DEA EN INFORMATIQUE

Sémantiques opérationnelles et domaines abstraits pour l'analyse statique de Java

Pollet, Isabelle

Award date:
1999

Facultés Universitaires Notre-Dame de la Paix
Namur
Institut d'Informatique

Sémantiques opérationnelles et domaines abstraits pour l'analyse statique de Java

Isabelle Pollet

Mémoire présenté en vue
de l'obtention du Diplôme
d'Etudes Approfondies (DEA)
en Informatique

Promoteur : B. Le Charlier

Année académique 1998-1999

Abstract

Object-oriented languages are difficult to implement efficiently, noticeably because the actual code to be executed by a method call is normally only determined at run-time (this is known as *dynamic dispatch*) and because an object-oriented program mainly consists of many small method declarations, most of which are built as small increments or modifications of others, according to the principle of inheritance. Therefore, there is a strong motivation to use static analyses techniques to derive semantic properties of programs allowing a compiler to specialize and optimize generated code. Static analyses are also useful to detect programming errors at compile-time rather than run-time.

In this thesis, we consider the static analysis of (a subset) of the programming language Java, according to the methodology of abstract interpretation. We achieve two essential preliminary steps to obtain a correct and powerful analyser for Java: first, we define a precise operational semantics for a well-defined subset of the language; second, we define an abstract domain to support an abstract semantics of the language. For pragmatic reasons, variants of the semantics and of the domain are proposed and discussed. We conclude with a discussion of further works and with some pointers to the literature.

Résumé

La détermination dynamique du code à exécuter lors d'un appel de méthode rend difficile l'implémentation efficace des langages orientés objet; d'autant plus que l'exploitation du principe de l'héritage conduit à programmer de courtes méthodes qui rajoutent un minimum de code aux méthodes de leur "super classe". Il est donc très intéressant de pouvoir dériver des propriétés sémantiques des programmes qui permettent au compilateur de spécialiser et d'optimiser le code généré. L'analyse statique permet également la détection d'erreurs à la compilation plutôt qu'à l'exécution.

Dans ce mémoire, nous abordons l'analyse statique d'un sous-langage de Java selon les méthodes de l'interprétation abstraite. Nous réalisons deux étapes préliminaires essentielles dans la construction d'un analyseur correct et puissant pour Java : nous définissons, d'une part, une sémantique opérationnelle précise pour un sous-ensemble de Java et, d'autre part, un domaine abstrait pouvant servir de base à l'élaboration d'une sémantique abstraite. Pour des raisons pratiques, nous proposons et discutons différentes variantes pour cette sémantique et de ce domaine. Nous concluons par la liste des tâches restant à réaliser et nous indiquons quelques références bibliographiques qui ont influencé notre travail.

Avant d'entrer dans le vif du sujet, je tiens à remercier mon promoteur, le professeur B. Le Charlier, qui même les jours les plus ensoleillés, a su rester disponible. Il est évident que, sans son aide, ce mémoire n'aurait jamais existé.

Je remercie également Christine, Christophe, Denis, Karl et Samuel pour ces heures passées à lire ... autre chose qu'une "bonne bande dessinée".

Table des matières

Introduction **9**

1 Aspects syntaxiques **11**

1.1 Quelques notations 12

1.1.1 Ensembles et éléments 12

1.1.2 Symboles classiques 12

1.2 Une première syntaxe abstraite 13

1.2.1 Ensembles atomiques 13

1.2.2 Désignateurs et expressions 13

1.2.3 Instructions 14

1.2.4 Types 14

1.2.5 Déclaration de méthode 15

1.2.6 Déclaration de constructeur 15

1.2.7 Définition de classe 16

1.2.8 Programme 17

1.3 Syntaxe abstraite et règles de typage 17

1.3.1 Syntaxe abstraite typée (SAT) 17

1.3.2 Environnements de types 19

1.3.3 Programme bien typé 25

1.4 Syntaxe abstraite et points de programme 31

1.4.1 Syntaxe abstraite "labelisée" (SAP) 31
1.4.2 Syntaxe abstraite bien "labelisée" 32
1.4.3 Exemple de "labelisation" 35
1.4.4 Quelques définitions supplémentaires 36

2 Sémantique opérationnelle **39**

2.1 Domaine d'interprétation 40
2.1.1 Valeurs 40
2.1.2 Environnement et store 41
2.1.3 Définition du domaine 42
2.1.4 Evaluation d'un désignateur 44
2.1.5 Evaluation d'une expression 46
2.1.6 Remarques sur la signification du domaine 47
2.2 Système de transitions 51
2.2.1 Présentation générale 51
2.2.2 Ensemble des états 52
2.2.3 Règles simples 53
2.2.4 Règles pour les appels 54
2.2.5 Règles pour les retours 59
2.2.6 Etat initial et règle de clôture 61
2.2.7 Remarque finale 61

3 Une seconde sémantique **63**

3.1 Domaine 64
3.1.1 Définition 64
3.1.2 Deux nouvelles fonctions 65
3.2 Système de transitions 66

3.2.1 Etats 66
3.2.2 Interprétation 67
3.2.3 Règles simples 68
3.2.4 Règles pour les appels 68
3.2.5 Règles pour les retours 70
3.2.6 Etat initial et règle de clôture 75
3.3 Lien entre les deux sémantiques 76
3.3.1 Approximation 76
3.3.2 Elimination du "bruit" 77

4 Domaine abstrait **81**

4.1 Domaine abstrait primitif 83
4.1.1 Exemple 1 : ensembles de types 83
4.1.2 Exemple 2 : types 83
4.2 Forme du domaine 84
4.2.1 Valeurs abstraites 84
4.2.2 Environnements et stores abstraits 85
4.2.3 Domaine abstrait 86
4.3 Fonction de concrétisation 87
4.4 Fonction d'abstraction 91
4.5 Equivalence intuitive 92
4.6 Préordre 94
4.7 Equivalence classique 99
4.8 Borne supérieure 102
4.8.1 Rappels 102
4.8.2 Exemples 102
4.9 Modification du domaine 107

4.9.1 Elargissement de $\mathbb{D}^{\#}$ 108

4.9.2 Ebauche d'un nouveau domaine 110

5 Domaine abstrait (version 2) **119**

5.1 Fonction de concrétisation 120

5.2 Fonction d'abstraction 121

5.3 Préordre 121

5.4 Borne supérieure 125

5.4.1 Quelques exemples 126

5.4.2 Quelques notations 129

5.4.3 Algorithme 131

5.4.4 Correction de l'algorithme 134

5.5 Illustration 140

Conclusion **143**

Bibliographie **147**

Introduction

Notre travail se situe dans le contexte de l'analyse statique de programmes. Plus précisément, nous tentons ici d'appliquer des méthodes d'interprétation abstraite à un langage du paradigme orienté objet, à savoir Java.

L'analyse statique de programmes prend son sens principalement dans deux domaines : la vérification et l'optimisation. Pour une étude plus détaillée de l'utilité de celle-ci dans le cadre spécifique de la programmation orientée objet, on se référera à [CDG].

Attardons-nous cependant sur quelques utilisations potentielles de l'analyse statique en O.O.

Du point vu de de l'optimisation, on constate que le style de programmation basé sur l'héritage et le polymorphisme propre à l'orienté objet introduit un "overhead" considérable dû à l'utilisation systématique d'appels de méthodes virtuelles dont la version réelle est déterminée seulement au moment de l'exécution. On peut chercher à remplacer cette détermination dynamique par une détermination statique basée sur une analyse intra- ou inter-procédurale du type des variables. Des études expérimentales ont montré que les gains d'efficacité peuvent être considérables (jusqu'à plusieurs ordres de grandeur [CDG]). D'autres optimisations intéressantes incluent la spécialisation des procédures polymorphiques (remplacement d'une version générique par plusieurs versions spécifiques) et "l'in-lining" consistant à remplacer les appels de méthodes par le code de celle-ci. La plupart de ces optimisations nécessitent une connaissance précise des types réels possibles des variables à l'exécution.

Si maintenant on se place dans un objectif de vérification, on peut partir de l'observation que le "casting" est une source importante d'erreurs à l'exécution (il est par ailleurs également une source d'inefficacité). Le "casting" permet au programmeur de prédire le type d'une variable à l'exécution sans que les règles normales de typage puissent garantir la correction de cette prédiction. Une analyse statique précise des types à l'exécution permet de détecter les erreurs de prédiction et/ou de confirmer la correction des prédictions. Dans un autre ordre d'idées, on peut également s'intéresser à la détection des variables locales non initialisées (le langage Java réalise par ailleurs des vérifications de ce type).

Nous nous intéressons ici à l'analyse statique des types des variables pour le langage Java. Ce choix s'explique par l'intérêt grandissant pour ce langage combiné au fait qu'il soit plus clairement défini que d'autres comme, par exemple, C++.

L'originalité de ce travail réside dans l'application systématique des principes de l'interprétation

abstraite à Java : en effet, notre méthode repose sur l'abstraction d'une sémantique opérationnelle réaliste du langage. La plupart des autres travaux existants (sauf, par exemple, les travaux d'Alain Deutsch [Deu94]) basent leurs analyses sur une description simplifiée ad hoc (i.e. non obtenue par interprétation abstraite) du langage (voir [CDG]). Ces approches se justifient par des considérations pragmatiques d'efficacité qui ne sont toutefois que partiellement convaincantes. Il n'y a en effet pas de démonstration du fait qu'une approche directe de l'analyse par interprétation abstraite soit définitivement inefficace.

Dans ce travail, nous entamons une recherche basée sur l'approche directe par interprétation abstraite en nous inspirant de travaux qui ont prouvé leur efficacité pour l'interprétation abstraite de Prolog (voir [LCVH94]). Il est intéressant de noter que les auteurs de ces travaux ont également d'abord reçu des commentaires négatifs de la communauté de programmation logique quant à la viabilité de leur approche. Mais son efficacité fut par la suite démontrée expérimentalement [LCVH94] et n'est plus mise en doute actuellement.

Nous espérons donc obtenir un succès analogue pour l'analyse de Java. Il faut noter aussi que nous n'essayons pas d'être aussi générique (i.e. indépendant du langage) que d'autres travaux (voir [CDG]) car le modèle opérationnel de Java est assez différent d'autres langages comme C++ et parce que nous désirons réaliser une modélisation précise de la sémantique du langage.

Il est évident que ce mémoire ne couvre qu'un préfixe du travail à réaliser : la construction complète d'un analyseur requiert beaucoup plus de temps que le temps imparti à la rédaction d'un "travail de fin d'études".

Ce travail poursuit deux objectifs : rédiger une sémantique opérationnelle d'un sous-langage "raisonnable" de Java et, à partir de celle-ci, poser les premiers jalons de l'analyseur de types.

Les deux premiers chapitres sont entièrement dédiés au premier objectif. Le chapitre 1 délimite le sous-langage étudié par la présentation d'une syntaxe abstraite correspondant à celui-ci. Il propose également un système de règles de vérification de type pour cette syntaxe. Le chapitre 2 est consacré quant à lui à la rédaction de la sémantique.

Les chapitres suivants s'insèrent naturellement dans le second objectif. Le chapitre 3 propose une version "remodelée" de la sémantique que nous estimons plus adaptée à l'analyse. Les chapitres 4 et 5 présentent deux domaines abstraits "duaux" et soulèvent quelques problèmes liés à ceux-ci.

Dans le développement que nous proposons, nous ne faisons pas de références à d'autres travaux. Cette carence s'explique simplement par le fait que le temps imparti à la préparation et à la rédaction de ce mémoire ne nous a permis ni d'approfondir la littérature relative à Java ni celle relative à l'interprétation abstraite. En ce qui concerne le langage Java, nous avons suivi essentiellement [LC99a] qui présente une définition concise d'un sous-ensemble restreint de Java; pour l'interprétation abstraite, nous nous sommes basés principalement sur les notes du cours de DEA [LC99b]. Une présentation succincte de recherches en interprétation abstraite ayant des liens avec notre travail est fournie à la fin de ce mémoire.

Chapitre 1

Aspects syntaxiques

Dans ce travail, un premier objectif est d'établir une sémantique d'un sous-langage de Java. Par conséquent, débuter par la délimitation de ce sous-langage s'impose : c'est ce que nous réalisons dans ce chapitre par l'intermédiaire d'une syntaxe abstraite.

Nous utilisons, en fait, plusieurs syntaxes abstraites. La première syntaxe (nommée **SA**) découle "directement" de la syntaxe de Java et tente de trouver un compromis entre les préoccupations suivantes : respect de la "philosophie", de la présentation et des possibilités du langage initial[1]; limitation du langage aux caractéristiques pertinentes pour l'analyse entamée; possibilité de construction et de rédaction d'une sémantique de taille et de complexité "raisonnables". Les deux dernières préoccupations entraînent bien sûr des réductions "drastiques" du langage.

La deuxième syntaxe (nommée **SAT**) est une légère variante de **SA**. Elle est introduite dans le seul but de définir la notion de "programme bien typé" au moyen de règles de déduction.

La troisième et dernière syntaxe (nommée **SAP**) introduit la notion de point de programme : elle traduit donc déjà l'optique "système de transitions" adoptée par la suite pour la définition de la sémantique.

Ce chapitre se structure comme suit. Avant tout, nous reprécisons quelques conventions et notations (usuelles) adoptées tout au long du chapitre. Nous entrons dans le vif du sujet en donnant la définition de la syntaxe **SA** et en rappelant, de manière informelle, quelques règles additionnelles élémentaires. Nous définissons alors la syntaxe **SAT** et la notion "d'environnement de types"; ces deux notions permettant elles-mêmes la définition de ce qu'est un "programme bien typé". Nous terminons par la définition de **SAP**, seule syntaxe utilisée dans les chapitres suivants.

[1] Remarquons qu'on ne démarre évidemment pas du langage Java lui-même mais d'une portion déjà restreinte de celui-ci présentée dans [LC99a].

1.1 Quelques notations

Nous employons, dans ce document, des notations usuelles pour les descriptions de syntaxe abstraite. Afin d'éviter toute confusion, reprécisons cependant quelques conventions adoptées et quelques symboles utilisés.

1.1.1 Ensembles et éléments

Les ensembles syntaxiques sont notés en italique avec une première lettre majuscule tandis que leurs éléments, également en italique, sont écrits complètement en minuscules.

En ce qui concerne les éléments de ces ensembles, nous tentons de leur donner des noms suffisament représentatifs pour ne pas devoir repréciser constamment à quel ensemble l'un ou l'autre composant appartient; ce qui s'avèrerait terriblement lourd dans la rédaction de certaines règles. Lors des descriptions de syntaxe, chaque élément porte le nom de l'ensemble auquel il appartient; dans les autres types de définitions, nous sommes souvent contraints d'utiliser des noms plus courts (les premières lettres de l'ensemble par exemple).

On rappelle les relations d'appartenance uniquement dans les situations pouvant porter à confusion.

1.1.2 Symboles classiques

Reprécisons quelques symboles utilisés :

- les "*" apposées en exposant symbolisent des séquences[2] (ainsi $type^*$ représente une séquence, peut-être vide, d'éléments de l'ensemble $Type$),
- les "+" apposés en exposant symbolisent des séquences non vides,
- des parenthèses sont ajoutées pour augmenter la lisibilité ou pour préciser la portée d'une "*" ou d'un "+",
- les crochets entourent des éléments facultatifs,
- les symboles terminaux sont imprimés en gras.

[2]Dans ce travail, nous amalgamons de manière tout à fait abusive les séquences et les n-uplets. Par exemple, le passage de l'ensemble $Type^*$ (qui peut être vu comme l'union disjointe des ensembles $Type^n$) à l'ensemble $Type^k$ est fait directement, sans aucune mention explicite au projecteur approprié.

1.2 Une première syntaxe abstraite

Dans cette section, nous fournissons la définition de la syntaxe **SA**. Nous donnons, en outre, au fur et à mesure et de manière informelle, quelques règles syntaxiques supplémentaires non directement traduites ou traductibles dans notre formalisme.

Remarquons que la plupart des contraintes habituellement énoncées à ce niveau, comme les règles concernant les arités des méthodes, seront décrites lors de la vérification des types. Les règles mentionnées ici sont en fait plutôt données à titre indicatif et n'ont aucune prétention d'exhaustivité.

1.2.1 Ensembles atomiques

Nous choisissons, tout à fait arbitrairement, de ne traiter que des valeurs "de base" entières et booléennes. De plus, la plupart des aspects relatifs à ces valeurs "de base" n'étant pas vraiment pertinents pour notre analyse, les définitions propres à ceux-ci resteront volontairement incomplètes.

Nous regroupons par conséquent les littéraux (entiers et booléens) dans un seul et même ensemble noté *Litt*.

Les ensembles de base "réellement" intéressants sont :

- $Nclasse$, ensemble des noms de classes,
- $Nvar$, ensemble des noms de variables,
- $Nmethode$, ensemble des noms de méthodes,
- $Nchamp$, ensemble des noms de champs.

Soulignons une première simplification majeure : nous éliminons toute notion de "package" en supposant que tous les packages sont rassemblés en un seul et même programme. Cette simplification en entraîne directement une autre qui intervient ici : on peut amalgamer les différentes dénominations des classes et des méthodes puisque les notions de "noms longs" et "noms courts" n'ont plus réellement de sens.

Comme d'habitude, ces ensembles atomiques sont supposés disjoints deux à deux.

1.2.2 Désignateurs et expressions

Intuitivement, un désignateur correspond à tout ce qui peut se trouver dans le membre de gauche d'une affectation tandis qu'une expression correspond à tout ce qui peut se trouver dans le membre de droite d'une affectation; du point de vue sémantique, on s'attend à attacher à un

désignateur une "adresse" tandis qu'on s'attend à attacher à une expression une "valeur" (nous repréciserons naturellement ces notions en temps utiles).

Nous distinguons deux types de désignateurs : les désignateurs "simples" correspondent à la notion évoquée au paragraphe précédent tandis que les "désignateurs d'instance" désignent, comme leur nom l'indique, des instances (des objets) et donc plutôt des "valeurs".

des ::= *nvar* | *nchamp* | *desinst nchamp*
desinst ::= **this** | **super** | *des*
expr ::= **null** | *litt* | *op expr** | *desinst*

Op désigne un ensemble d'opérateurs prédéfinis sur les valeurs de base sur lesquels nous ne nous attarderons pas. On suppose juste que les arités de ces opérateurs sont respectées.

Une expression désigne donc soit une valeur de base soit un objet. L'expresion **null** désigne un objet "qui n'existe pas" (pas d'instance créée).

this désigne l'instance courante et **super**, si cela a un sens, l'instance "père" de l'instance courante (dans la relation d'héritage définie par le programme).

Remarquons que l'on s'attendrait à trouver, dans les expressions, des appels de constructeurs et de fonctions. On préfère, en fait, "transporter" ceux-ci au niveau des instructions et ce, en anticipant sur la suite, pour éviter les effets de bord dans l'évaluation des expressions. Schématiquement, un appel de fonction *appel* situé dans une expression *expr* est remplacé par une "nouvelle variable" y dans *expr* sur laquelle on a, au préalable, effectué l'affectation $y :=$ *appel*.

1.2.3 Instructions

On retrouve les instructions classiques d'affectation, de test et de boucle, ainsi que l'instruction de fin de méthode **return**. Les appels de méthodes ont une forme simplifiée : on ne peut pas retrouver n'importe quel désignateur d'instance comme paramètre principal, mais seulement une variable, **this** ou **super**.

instr ::= **affect** *des expr* | **if** *expr instr* instr** | **proc** *appel* | **return** [*expr*]
| **fonc** *nvar appel* | **constr** *nvar nclasse expr** | **while** *expr instr**
appel ::= **this** *nmethode expr** | **super** *nmethode expr** | *nvar nmethode expr**

1.2.4 Types

Un type est soit un type de base, soit un nom de classe. Nous ajoutons le type "fictif" **void** pour permettre une syntaxe commune pour les procédures et les fonctions.

type ::= **int** | **bool** | *nclasse* | **void**

1.2.5 Déclaration de méthode

Nous distinguons deux types de déclarations de méthodes : les déclarations de méthodes abstraites et les déclarations de méthodes concrètes. Une déclaration de méthode abstraite consiste en une simple "en-tête" tandis qu'une déclaration de méthode concrète comporte une "en-tête" mais aussi des déclarations de variables locales[3] et un corps de méthode.

$$\begin{aligned} declmethode \ ::= \ & type\ nmethode\ (type\ nvar)^* \\ & \mid type\ nmethode\ (type\ nvar)^*\ (type\ nvar)^*\ instr^+ \end{aligned}$$

Toutes les variables mentionnées dans l'en-tête et dans les déclarations locales sont distinctes. Le seul type pouvant être égal à **void** est le type résultat (à savoir le premier type mentionné dans la déclaration).

S'il s'agit d'une fonction (i.e. si le type résultat est différent de **void**), toutes les intructions **return** du corps de la méthode sont suivies d'une expression; s'il s'agit d'une procédure, toutes les instructions **return** sont sans expression.

Nous supposons que le corps d'une méthode se termine toujours par une instruction **return**.

Remarquons que nous laissons tomber tout ce qui est relatif aux attributs d'accessibilité, ainsi que tout ce qui est dit "static". Nous laissons également tomber les notions du type "final" pour des raisons d'homogénéité de traitement (ces dernières peuvent cependant avoir un lien direct avec l'analyse).

Définition 1.1 (Signature d'une déclaration de méthode)

La signature d'une déclaration de méthode est composée du nom de la méthode et des types des paramètres, exception faite du paramètre résultat, i.e. la signature de la déclaration de méthode

$tr\ methode\ (t_1 v_1 ... t_n v_n)\ [(tt_1 w_1 ... tt_m w_m)\ (i_1 ... i_k)]$

est

$methode\ (t_1 ... t_n)$.

Soulignons que, dans notre définition, la signature d'une méthode ne reprend ni le type du résultat ni le type de l'objet sur lequel s'applique la méthode.

1.2.6 Déclaration de constructeur

On distingue trois types de déclarations de constructeurs :

- les déclarations mentionnant le symbole terminal **prem** correspondent à des classes sans

[3] Remarquons qu'on désigne par le terme *variable* uniquement les paramètres et les variables locales des méthodes et non les noms de champs de l'instance courante.

"père", il s'agit de "premiers constructeurs" dans le sens où ces constructeurs ne font pas appel à un autre constructeur de la classe;

- les déclarations mentionnant le symbole terminal **super** correspondent à des classes qui héritent d'une autre classe, il s'agit de "premiers constructeurs" dans le sens où ces constructeurs ne font pas appel à un autre constructeur de la classe;
- les déclarations mentionnant le symbole terminal **this** correspondent aussi bien à des classes sans père qu'à des classes qui héritent, il ne s'agit pas de "premiers constructeurs".

$$\begin{array}{lll} \textit{declconstr} & ::= & \textit{nclasse}\ (\textit{type nvar})^*\ (\textit{type nvar})^*\ \textbf{prem}\ \textit{instr}^+ \\ & & \mid \textit{nclasse}\ (\textit{type nvar})^*\ (\textit{type nvar})^*\ \textbf{super}\ \textit{expr}^*\ \textit{instr}^+ \\ & & \mid \textit{nclasse}\ (\textit{type nvar})^*\ (\textit{type nvar})^*\ \textbf{this}\ \textit{expr}^*\ \textit{instr}^+ \end{array}$$

Définition 1.2 (Signature d'une déclaration de constructeur)

La signature d'une déclaration de constructeur est composée du nom du constructeur et des types des paramètres.
Par exemple la signature de la déclaration $(nc\ (t_1 v_1, .., t_n v_n)\ \textbf{prem}\ (i_1 ... i_k))$ *est tout simplement* $(nc\ (t_1 ... t_n))$.

De nouveau, toutes les variables mentionnées dans l'en-tête et toutes les variables locales sont distinctes.

Tous les types mentionnés sont différents de **void**.

Toutes les instructions **return** sont sans expression et la dernière instruction du corps du constructeur est une instruction **return**.

1.2.7 Définition de classe

Une définition de classe se compose du nom de la classe, d'une éventuelle classe "père", d'une suite de déclarations de champs, d'une suite de déclarations de méthodes et d'une suite de déclarations de constructeurs.

$$\begin{array}{lll} \textit{defclass} & ::= & \textit{nclasse}\ [\textbf{extend}\ \textit{nclasse}]\ \textit{declchamp}^*\ \textit{declmethode}^*\ \textit{declconstr}^+ \\ \textit{declchamp} & ::= & \textit{type nchamp expr} \end{array}$$

Si le symbole **extend** figure dans la définition, cela signifie que la classe "étend" une autre classe. C'est l'ensemble des informations de ce type qui construit la relation d'héritage au sein d'un programme.

Citons quelques règles élémentaires qui doivent être vérifiées pour chaque déclaration de classe :

- tous les noms de champ sont distincts;

- il n'existe pas deux déclarations de méthode ou de constructeur de même signature;
- une classe ne s'étend pas elle-même;
- il existe au moins un "premier constructeur", i.e. si la classe n'a pas de "père" tous les constructeurs sont de la forme **this** ou **prem** et il existe au moins un constructeur de la forme **prem**; si la classe étend une autre classe, tous les constructeurs sont de la forme **this** ou **super** et il existe au moins un constructeur de la forme **super**.

1.2.8 Programme

Un programme se définit simplement comme une suite non vide de définitions de classe.

$$prog \ ::= \ defclass^{+}$$

On exige qu'il n'existe pas deux définitions de classe pour un même nom de classe.

On suppose qu'il existe une et une seule définition de classe comportant une déclaration de méthode de signature *main ()*.

La figure 1.1 fournit un résumé de la syntaxe **SA** présentée tout au long de cette section.

1.3 Syntaxe abstraite et règles de typage

Cette section a pour but de définir quand un programme de **SA** est bien typé (on supposera en effet par la suite que tous les programmes manipulés sont bien typés). Pour ce faire, nous procéderons selon le schéma suivant.

1. Nous introduisons une variante légèrement enrichie de **SA** : **SAT**.
2. Nous définissons un concept "d'environnement de types", indépendamment de toute notion de programme.
3. Nous définissons ce qu'est un programme de **SAT** bien typé via un système de règles de déduction manipulant des environnements de types. Les règles de typage sont présentées dans une optique "bottom-up".

1.3.1 Syntaxe abstraite typée (SAT)

Comme on l'a déjà signalé, **SAT** est une version de **SA** légèrement enrichie. On ajoute simplement des types au niveau des expressions, des désignateurs et des appels.

Cette information de "typage" est une information "factice" introduite pour faciliter l'expression de la définition de programme bien typé (mais aussi l'expression de certaines règles sémantiques).

Ensembles atomiques

litt
nclasse, *nvar*, *nmethode*, , *nchamp*

Ensembles construits

prog	::=	$defclass^+$
defclass	::=	*nclasse* [**extend** *nclasse*] *declchamp** *declmethode** $declconstr^+$
type	::=	**int** | **bool** | *nclasse* | **void**
declchamp	::=	*type nchamp expr*
declmethode	::=	*type nmethode* (*type nvar*)* | *type nmethode* (*type nvar*)* (*type nvar*)* $instr^+$
declconstr	::=	*nclasse* (*type nvar*)* (*type nvar*)* **prem** $instr^+$ | *nclasse* (*type nvar*)* (*type nvar*)* **super** *expr** $instr^+$ | *nclasse* (*type nvar*)* (*type nvar*)* **this** *expr** $instr^+$
instr	::=	**affect** *des expr* | **if** *expr instr** *instr** | **proc** *appel* | **return** [*expr*] | **fonc** *nvar appel* | **constr** *nvar nclasse expr** | **while** *expr instr**
appel	::=	**this** *nmethode expr** | **super** *nmethode expr** | *nvar nmethode expr**
des	::=	*nvar* | *nchamp* | *desinst nchamp*
desinst	::=	**this** | **super** | *des*
expr	::=	**null** | *litt* | *op expr** | *desinst*

Figure 1.1: Résumé de la syntaxe abstraite **SA**

Elle est en fait, dans le cas de programmes bien typés, totalement redondante et devrait alors s'obtenir de manière "constructive" à partir des environnements de types (optique de définition inverse à celle adoptée).

Une autre modification par rapport à **SA** se situe au niveau de la définition de l'ensemble *Type* lui-même. En effet, celui-ci possède un élément supplémentaire, noté **bot**, qui est introduit pour pouvoir "typer" l'expression **null**.

La définition de **SAT** est fournie dans la figure 1.2

1.3.2 Environnements de types

La notion d'environnement de types, que nous définissons pas à pas dans cette section, est la clé de voûte de notre définition de programme bien typé. Intuitivement, l'environnement de types d'un programme doit d'une part être cohérent avec les différentes définitions de classes du programme et d'autre part reprendre toute l'information de celui-ci nécessaire au typage de n'importe quelle expression du programme.

Nous choisissons de définir cette notion indépendamment de tout programme et d'établir le lien ultérieurement. Pour faciliter l'intuition, nous donnons cependant quelques explications en termes de morceaux de programme.

1.3.2.1 Relation d'extension

Il s'agit de traduire les éventuelles indications d'extension au niveau des définitions de classes. Il est évident que pour qu'un programme soit acceptable, cette relation ne peut être quelconque puisqu'on ne doit pas pouvoir permettre de "cycle" au niveau de l'extension.

Définition 1.3 (Relation d'extension)

Une relation d'extension, notée π, *est une relation fonctionnelle anti-réflexive sur* $Nclasse \times Nclasse$ *telle que*

$$\forall nc \in Nclasse, (nc, nc) \notin \pi^+$$

où π^+ *dénote la fermeture transitive de* π.

Il est évident que comme $Nclasse$ est fini, toute relation d'extension est bien fondée.

Attendu que les relations d'extension sont des fonctions, on utilisera pour celles-ci tantôt des notations relationnelles tantôt des notations fonctionnelles.

Ensembles atomiques

litt

nclasse, *nvar*, *nmethode*, , *nchamp*

Ensembles construits

prog	::=	$defclass^+$
defclass	::=	*nclasse* [**extend** *nclasse*] *declchamp** *declmethode** $declconstr^+$
type	::=	**bot** | **int** | **bool** | *nclasse* | **void**
declchamp	::=	*type nchamp expr*
declmethode	::=	*type nmethode* (*type nvar*)* | *type nmethode* (*type nvar*)* (*type nvar*)* $instr^+$
declconstr	::=	*nclasse* (*type nvar*)* (*type nvar*)* **prem** $instr^+$ | *nclasse* (*type nvar*)* (*type nvar*)* **super** *expr** $instr^+$ | *nclasse* (*type nvar*)* (*type nvar*)* **this** *expr** $instr^+$
instr	::=	**affect** *des expr* | **if** *expr instr** *instr** | **proc** *appel* | **return** [*expr*] | **fonc** *nvar appel* | **constr** *nvar nclasse expr** | **while** *expr instr**
appel	::=	*type* **this** *nmethode expr** | *type* **super** *nmethode expr** | *type nvar nmethode expr**
des	::=	*type nvar* | *type nchamp* | *type desinst nchamp*
desinst	::=	*type* **this** | *type* **super** | *des*
expr	::=	*type* **null** | *type litt* | *type op expr** | *desinst*

Figure 1.2: Définition de **SAT**

1.3.2.2 Relation de spécialisation

Comme son nom l'indique, cette notion traduit les liens de spécialisation entre types induits par un programme.

Définition 1.4 (Relation de spécialisation)

Etant donné une relation d'extension π, on définit la relation de spécialisation induite, notée $\preceq_\pi$, par

- $\preceq_\pi \subseteq Type \times Type$,
- $t \preceq_\pi t' \Leftrightarrow (t = \mathbf{bot} \wedge t' \in Nclasse + \{\mathbf{bot}\})$
  $\vee\ (t = t' \wedge t, t' \in \{\mathbf{bool}, \mathbf{int}\})$
  $\vee\ (t, t') \in \pi^*$.

π^* dénote la fermeture réflexo-transitive de π.

1.3.2.3 Environnement de types pour les champs

Définition 1.5 (Environnement primitif pour les champs)

Un environnement primitif pour les champs est une fonction de la forme

$$\epsilon^0_{ch} : Nclasse \longrightarrow Nchamp \rightarrowtail Type \setminus \{\mathbf{bot}, \mathbf{void}\}.$$

Intuitivement, le domaine de $\epsilon^0_{ch}\ nc$ correspond à l'ensemble des champs déclarés dans la définition de la classe nc et $\epsilon^0_{ch}\ nc\ nch$ dénote le type déclaré pour le champ nch.

Une relation d'extension permet d'étendre le domaine de définition d'un tel environnement. C'est l'environnement résultant de cette extension que nous nommerons "environnement de types pour les champs".

Définition 1.6 (Environnement de types pour les champs)

Etant donnés un environnement primitif pour les champs ϵ^0_{ch} et une relation d'extension π, on définit la fonction suivante :

$$\epsilon_{ch} : Nclasse \longrightarrow Nchamp \rightarrowtail Type \setminus \{\mathbf{bot}, \mathbf{void}\}$$

si $nch \in dom(\epsilon^0_{ch}\ nc)$
alors $\epsilon_{ch}\ nc\ nch = \epsilon^0_{ch}\ nc\ nch$
sinon **si** $nc \in dom(\pi)$
alors $\epsilon_{ch}\ nc\ nch = \epsilon_{ch}\ \pi(nc)\ nch$
sinon $nch \notin dom(\epsilon_{ch}\ nc)$

Cette définition a bien un sens puisqu'il s'agit d'une définition inductive le long de la relation π et que cette relation est bien fondée.

Dorénavant, lorsque nous fournirons, pour une fonction partielle, une définition similaire à la définition 1.6, nous supposerons que tous les cas non explicitement couverts par la définition correspondent à des points n'appartenant pas au domaine de la fonction en cours de définition (ce qui revient, dans le cas présent, à omettre la dernière ligne).

1.3.2.4 Environnement de types pour les méthodes

Définition 1.7 (Environnement primitif pour les méthodes)

Un environnement primitif pour les méthodes est une fonction de la forme

$$\epsilon_m^0 : Nclasse \longrightarrow Nmethode \longrightarrow (Type \setminus \{\mathbf{bot}, \mathbf{void}\})^* \rightarrowtail Type \setminus \{\mathbf{bot}\}.$$

Intuitivement, $\epsilon_m^0\ nc\ m\ (t_1...t_n)$ dénote le type résultat de la méthode de signature $m\ (t_1...t_n)$ déclarée dans la classe nc.

De nouveau, une relation d'extension permet d'étendre le domaine de définition d'un tel environnement.

Définition 1.8 (Environnement étendu pour les méthodes)

Etant donnés un environnement primitif pour les méthodes ϵ_m^0 et une relation d'extension π, on définit la fonction suivante :

$$\epsilon_m^1 : Nclasse \longrightarrow Nmethode \longrightarrow (Type \setminus \{\mathbf{bot}, \mathbf{void}\})^* \rightarrowtail Type \setminus \{\mathbf{bot}\}$$

si $(t_1..t_n) \in dom(\epsilon_m^0\ nc\ m)$
alors $\epsilon_m^1\ nc\ m\ (t_1...t_n) = \epsilon_m^0\ nc\ m\ (t_1...t_n)$
sinon si $nc \in dom(\pi)$
alors $\epsilon_m^1\ nc\ m\ (t_1...t_n) = \epsilon_m^1\ \pi(nc)\ m\ (t_1...t_n)$

Là encore, il s'agit d'une définition par induction le long de la relation bien fondée π.

Pour aboutir à ce que nous nommerons "environnement de types pour les méthodes", il nous faut encore réaliser une extension du domaine de la fonction ϵ_m^1. En effet, si la première extension incorpore à l'environnement la relation d'héritage au niveau de l'objet lui-même, la seconde extension traite l'héritage au niveau des paramètres effectifs d'un appel de méthode.

Définition 1.9 (Environnement de types pour les méthodes)

Etant donnés un environnement primitif pour les méthodes ϵ_m^0 et une relation d'extension π, on peut construire la relation de spécialisation $\preceq_\pi$ et l'environnement étendu ϵ_m^1. On définit alors l'environnement de types pour les méthodes ϵ_m comme suit

$$\epsilon_m : Nclasse \longrightarrow Nmethode \longrightarrow (Type \setminus \{\textbf{void}\})^* \nrightarrow Type \setminus \{\textbf{void}\}$$

si $(t_1..t_n) \in dom(\epsilon_m^1\ nc\ m)$
alors $\epsilon_m\ nc\ m\ (t_1...t_n) = \epsilon_m^1\ nc\ m\ (t_1...t_n)$
sinon si $(\tau_1...\tau_n) = \min\{(tt_1...tt_n) \in dom(\epsilon_m^1\ nc\ m) \mid \forall i : 1 \leq i \leq n : t_i \preceq_\pi tt_i)\}$
alors $\epsilon_m\ nc\ m\ (t_1...t_n) = \epsilon_m^1\ nc\ m\ (\tau_1...\tau_n)$

1.3.2.5 Environnement de types pour les constructeurs

Définition 1.10 (Environnement primitif pour les constructeurs)

Un environnement primitif pour les constructeurs est une fonction de la forme

$$\epsilon_c^0 : Nclasse \longrightarrow (Type \setminus \{\textbf{bot}, \textbf{void}\})^* \nrightarrow Nclasse$$

qui vérifie

$$\forall (t_1...t_n) \in dom(\epsilon_c^0\ nc),\ \epsilon_c^0\ nc\ (t_1...t_n) = nc.$$

On garde pour les constructeurs une fonction de la même forme que pour les champs et les méthodes par souci d'homogénéité, toutefois ce qui nous intéresse ici est moins le résultat de la fonction, qui est constant, que son domaine.

Intuitivement, $\epsilon_c^0\ nc\ (t_1...t_n) = nc$ si et seulement si il existe une déclaration de constructeur de signature $nc\ (t_1...t_n)$ dans la classe nc.

Il est évident qu'incorporer une relation d'extension dans cet environnement n'aurait pas de sens : les constructeurs sont propres à chaque classe (et nous supposons, dans notre syntaxe, que toutes les déclarations de constructeurs sont explicites).

Par contre, comme pour les méthodes, il semble souhaitable que l'environnement tienne compte de l'héritage au niveau des paramètres.

Définition 1.11 (Environnement de types pour les constructeurs)

Etant donnés un environnement primitif pour les constructeurs ϵ_c^0 et une relation d'extension π, on peut construire la relation de spécialisation $\preceq_\pi$ et définir l'environnement de types pour les constructeurs ϵ_c comme suit

$$\epsilon_c : Nclasse \longrightarrow (Type \setminus \{\textbf{void}\})^* \not\longrightarrow Nclasse$$

si $(t_1..t_n) \in dom(\epsilon_c^0\ nc)$
alors $\epsilon_c\ nc\ (t_1...t_n) = nc$
sinon si $\{(tt_1...tt_n) \in dom(\epsilon_c^0\ nc) \mid \forall i : 1 \leq i \leq n : t_i \preceq_\pi tt_i)\}$ *admet un minimum*
alors $\epsilon_c\ nc\ (t_1...t_n) = nc$

1.3.2.6 Environnement global et environnement local

Un environnement global regroupe simplement toutes les informations des différents environnements présentés au cours de cette section.

Un environnement global primitif correspond à toute l'information "brute" relative aux types que l'on peut extraire d'un programme (bien typé) tandis que l'environnement global intègre plus directement (de manière cohérente avec l'environnement primitif) toutes les informations relatives à l'héritage.

Un environnement local est simplement un environnement global enrichi par une information propre au corps d'une méthode : les types des variables de la méthode.

Définition 1.12 (Environnement global de types primitif)

Un environnement global de types primitif est un quadruplet $(\epsilon_{ch}^0, \epsilon_m^0, \epsilon_c^0, \pi)$ où

- *ϵ_{ch}^0 est un environnement primitif pour les champs,*
- *ϵ_m^0 est un environnement primitif pour les méthodes,*
- *ϵ_c^0 est un environnement primitif pour les constructeurs,*
- *π est une relation d'extension.*

Définition 1.13 (Environnement global de types)

Etant donné un environnement global de types primitif $(\epsilon^0_{ch}, \epsilon^0_m, \epsilon^0_c, \pi)$, *on définit l'environnement de types* $(\epsilon_{ch}, \epsilon_m, \epsilon_c, \pi, \preceq_\pi)$ *où*

- ϵ_{ch} *est l'environnement de types pour les champs construit à partir de* ϵ^0_{ch} *et de* π,
- ϵ_m *est l'environnement de types pour les méthodes construit à partir de* ϵ^0_m *et de* π,
- ϵ_c *est l'environnement de types pour les constructeurs construit à partir de* ϵ^0_c *et de* π,
- $\preceq_\pi$ *est la relation de spécialisation induite par* π.

Définition 1.14 (Environnement local de types (primitif))

Un environnement local de types (primitif) est un quintuplet $(\epsilon^0_{ch}, \epsilon^0_m, \epsilon^0_c, \pi, \epsilon_v)$ *où*

- $(\epsilon^0_{ch}, \epsilon^0_m, \epsilon^0_c, \pi)$ *est un environnement global de types (primitif),*
- ϵ_v *est une fonction de la forme*

$$\epsilon_v : Nvar + \{\mathbf{this}\} \rightarrowtail\!\!\!\!\!\!\rightarrow Type \setminus \{\mathbf{bot}, \mathbf{void}\}$$

Par la suite, nous omettrons souvent de mentionner explicitement s'il s'agit d'environnement primitif ou non; nous signalerons la distinction uniquement au niveau des notations : ainsi, ϵ^0_m désignera un environnement primitif pour les méthodes et ϵ_m l'environnement de types pour les méthodes construit à partir de celui-ci et d'une relation d'extension π (non toujours explicitement mentionnée).

1.3.3 Programme bien typé

Dans cette section, nous présentons les différentes "règles de typage". Nous procédons selon une optique "bottom-up" : partant des plus "petits" ensembles constituants de **SAT**, nous remontons vers les plus "grands". C'est au niveau du dernier ensemble, à savoir *Prog*, que la jonction entre programme et environnement de types devient effective.

1.3.3.1 Désignateur et expression bien typés

Soit un environnement local de types $\epsilon = (\epsilon^0_{ch}, \epsilon^0_m, \epsilon^0_c, \pi, \epsilon_v)$; la notation

$$(\epsilon, E) \vdash e$$

se lit "e est un élément de E bien typé relativement à ϵ".

Les règles concernant le typage des expressions manipulant des "valeurs" de base restent partiellement "non détaillées" : dans ce qui suit, $\tau(litt)$ dénote simplement le type du littéral $litt$ et τ_{op} une fonction prédéfinie de signature $Type^* \nrightarrow Type$.

$$(\epsilon, Expr) \vdash \textbf{bot null}$$

$$(\epsilon, Expr) \vdash \tau(litt)\ litt$$

$$(\epsilon, Expr) \vdash \tau_{op}(t_1...t_n)\ (op(t_1e_1...t_ne_n))$$

Si une expression est un désignateur d'instance, on se réfère simplement aux règles sur les désignateurs d'instance.

$$\frac{(\epsilon, Desinst) \vdash desinst}{(\epsilon, Expr) \vdash desinst}$$

Le type statique de **this** doit être le type repris dans l'environnement de type pour les variables et il doit évidemment s'agir d'un nom de classe.

$$\frac{\epsilon_v(\textbf{this}) \in Nclasse}{(\epsilon, Desinst) \vdash \epsilon_v(\textbf{this})\ \textbf{this}}$$

super ne peut être bien typé que si **this** correspond, dans l'environnement de types pour les variables, à une classe étendue (i.e. appartenant au domaine[4] de π).

$$(\epsilon, Desinst) \vdash \pi(\epsilon_v\textbf{this})\ \textbf{super}$$

Un désignateur bien typé est un désignateur d'instance bien typé si "son type" est un nom de classe[5].

$$\frac{(\epsilon, Des) \vdash t\ des \quad t \in Nclasse}{(\epsilon, Desinst) \vdash t\ des}$$

Une variable est bien typée si son type correspond au type précisé dans l'environnement de types pour les variables.

$$(\epsilon, Des) \vdash \epsilon_v(nvar)\ nvar$$

Un nom de champ est bien typé si son type correspond au type fourni dans l'environnement de types pour les champs à partir du type de **this** dans l'environnement local.

$$(\epsilon, Des) \vdash (\epsilon_{ch}(e_v\textbf{this})\ nch)\ nch$$

[4]Remarquons, au niveau des notations, que si f est une fonction partielle, dés que l'on emploie la notation $f(x)$, on sous-entend que x appartient au domaine de f.

[5]Remarquons, du point de vue des notations, que, dans cette règle, *des* ne symbolise pas un élément de *Des* mais la partie restant à un tel élément quand on lui ôte l'annotation de type. On procèdera souvent de la sorte par la suite.

Un désignateur suivi d'un nom de champ est bien typé si son type correspond au type fourni dans l'environnement de types pour les champs à partir du type du désignateur.

$$\frac{(\epsilon, Desinst) \vdash t\ des}{(\epsilon, Des) \vdash (\epsilon_{ch}\ t\ nch)\ (t\ des\ nch)}$$

1.3.3.2 Instruction bien typée

Soit un environnement local de types ϵ et tr un élément de $Type \setminus \{\mathbf{bot}\}$, la notation

$$(\epsilon, E, tr) \vdash e$$

se lit "e est un élément de E bien typé relativement à ϵ pour le type résultat tr". Le type tr correspond au type de la méthode où se trouve le morceau de programme "en cours de traitement". Cette information, quelque peu artificielle, n'est pas disponible dans ϵ et est ajoutée pour pouvoir, au niveau des instructions, vérifier l'adéquation du type des expressions associées aux instructions **return**.

Une affectation est bien typée si le type de l'expression est une spécialisation du type du désignateur.

$$\frac{\begin{array}{l}(\epsilon, Des) \vdash t\ des \\ (\epsilon, Expr) \vdash t'\ expr \\ t' \preceq_\pi t\end{array}}{(\epsilon, Instr, tr) \vdash (t\ des)\ (t'\ expr)}$$

Une instruction de test est bien typée si son expression est typée **bool** et si les deux alternatives du test sont composées d'instructions bien typées.

$$\frac{\begin{array}{l}(\epsilon, Instr, tr) \vdash i_1, ..., i_k \\ (\epsilon, Instr, tr) \vdash j_1, ..., j_n \\ (\epsilon, Expr) \vdash (\mathbf{bool}\ expr)\end{array}}{(\epsilon, Instr, tr) \vdash \mathbf{if}\ (\mathbf{bool}\ expr)\ (i_1 ... i_k)\ (j_1 ... j_n)}$$

De même, une instruction **while** est bien typée si son expression est typée **bool** et si les instructions du corps de la boucle sont bien typées.

$$\frac{\begin{array}{l}(\epsilon, Instr, tr) \vdash i_1, ..., i_k \\ (\epsilon, Expr) \vdash (\mathbf{bool}\ expr)\end{array}}{(\epsilon, Instr, tr) \vdash \mathbf{while}\ (\mathbf{bool}\ expr)\ (i_1 ... i_k)}$$

Une instruction **return** est bien typée si le type de son expression est une spécialisation du type résultat.

$$(\epsilon, Instr, \mathbf{void}) \vdash \mathbf{return}$$

$$\frac{(\epsilon, Expr) \vdash (t\ expr) \quad t \preceq_\pi tr}{(\epsilon, Instr, tr) \vdash \textbf{return}\ (t\ expr)}$$

Un appel (dans le sens élément de l'ensemble *Appel*) constitue une instruction bien typée s'il s'agit bien d'un appel de procédure, ce que l'on modélise ici en typant l'appel par **void**.

$$\frac{(\epsilon, Appel) \vdash \textbf{void}\ appel}{(\epsilon, Instr, tr) \vdash \textbf{proc void}\ appel}$$

On construit un "appel de fonction" bien typé à partir d'une variable et d'un appel dont le type et une spécialisation du type de la variable.

$$\frac{(\epsilon, Appel) \vdash t\ appel \quad t \preceq_\pi \epsilon_v(nvar)}{(\epsilon, Instr, tr) \vdash \textbf{fonc}\ nvar\ (t\ appel)}$$

Pour qu'un appel de constructeur soit bien typé, il faut qu'il existe dans la classe un constructeur correspondant aux types des paramètres et que la classe soit une spécialisation du type de la variable sur laquelle s'effectue l'appel.

$$\frac{(\epsilon, Expr) \vdash t_1e_1, ..., t_ne_n \quad (t_1...t_n) \in dom(\epsilon_c\ nc) \quad nc \preceq_\pi \epsilon_v(nvar)}{(\epsilon, Instr, tr) \vdash \textbf{constr}\ nvar\ nc\ (t_1e_1...t_ne_n)}$$

Pour qu'un appel (de méthode) soit bien typé, il faut qu'il existe dans la classe (type de l'objet auquel s'applique l'appel) une méthode correspondant à la signature déterminée par le nom de la méthode et le type des paramètres.

$$\frac{(\epsilon, Expr) \vdash t_1e_1, ..., t_ne_n \quad x \in Nvar + \{\textbf{this}\} \quad \epsilon_m\ \epsilon_v(x)\ methode\ (t_1...t_n) = t}{(\epsilon, Appel) \vdash t\ x\ methode\ (t_1e_1...t_1e_n)}$$

$$\frac{(\epsilon, Expr) \vdash t_1e_1, ..., t_ne_n \quad \epsilon_m\ \pi(\epsilon_v(\textbf{this}))\ methode\ (t_1...t_n) = t}{(\epsilon, Appel) \vdash t\ \textbf{super}\ methode\ (t_1e_1...t_1e_n)}$$

1.3.3.3 Déclaration de méthode bien typée

Jusqu'à présent, les règles de typage s'expriment relativement à un environnement local (éventuellement accompagné d'un type résultat). Les règles correspondant aux différentes déclarations s'expriment quant à elles relativement à un environnement global auquel on adjoint le nom de la classe "en cours de traitement".

Soit un environnement global de types $\epsilon = (\epsilon^0_{ch}, \epsilon^0_m, \epsilon^0_c, \pi)$ et un nom de classe nc; la notation

$$(\epsilon, E, nc) \vdash e$$

se lit "e est un élément de E bien typé relativement à ϵ et à la classe nc".

On note $\perp_v$ la fonction de signature $Nvar + \{\mathbf{this}\} \rightarrowtail\!\!\!\!\!\!\rightarrow Type$ et de domaine vide.

Une déclaration de méthode concrète est bien typée si les instructions du corps de la méthode sont bien typées dans l'environnement local déduit des déclarations de types données pour les variables. Une déclaration de méthode abstraite est toujours bien typée.

$$(\epsilon, Declmethode, nc) \vdash tr\ methode\ (t_1 v_1 ... t_n v_n)$$

$$\frac{\begin{array}{l}\epsilon_v = \perp_v[\mathbf{this}/nc, v_1/t_1, ..., v_n/t_n, w_1/tt_1, ..., w_m/tt_m] \\ ((\epsilon, \epsilon_v), Instr, tr) \vdash i_1, ..., i_{k+1} \\ i_{k+1} = \mathbf{return}\ [expr]\end{array}}{(\epsilon, Declmethode, nc) \vdash tr\ methode\ (t_1 v_1 ... t_n v_n)\ (w_1 tt_1 ... w_m tt_m)\ (i_1 ... i_{k+1})}$$

1.3.3.4 Déclaration de constructeur bien typée

Comme pour les déclarations de méthodes, une déclaration de constructeur est bien typée si les instructions du corps du constructeur sont bien typées dans l'environnement local déduit des déclarations de types données pour les variables. Il faut en outre vérifier que l'éventuel appel initial à un autre constructeur est "bien typé".

$$\frac{\begin{array}{l}\epsilon_v = \perp_v[\mathbf{this}/nc, v_1/t_1, ..., v_n/t_n, w_1/tt_1, ..., w_m/tt_m] \\ ((\epsilon, \epsilon_v), Instr, \mathbf{void}) \vdash i_1, ..., i_{k+1} \\ i_{k+1} = \mathbf{return}\end{array}}{(\epsilon, Declconstr, nc) \vdash nc\ (t_1 v_1 ... t_n v_n)\ (w_1 tt_1 ... w_m tt_m)\ \mathbf{prem}\ (i_1 ... i_{k+1})}$$

$$\frac{\begin{array}{l}\epsilon_v = \perp_v[\mathbf{this}/nc, v_1/t_1, ..., v_n/t_n, w_1/tt_1, ..., w_m/tt_m] \\ ((\epsilon, \epsilon_v), Expr) \vdash \tau_1 e_1, ..., \tau_l e_l \\ (\tau_1 ... \tau_l) \in dom(\epsilon_c(nc)) \\ ((\epsilon, \epsilon_v), Instr, \mathbf{void}) \vdash i_1, ..., i_{k+1} \\ i_{k+1} = \mathbf{return}\end{array}}{(\epsilon, Declconstr, nc) \vdash nc\ (t_1 v_1 ... t_n v_n)\ (w_1 tt_1 ... w_m tt_m)\ \mathbf{this}\ (\tau_1 e_1 ... \tau_l e_l)\ (i_1 ... i_{k+1})}$$

$$\frac{\begin{array}{l}\epsilon_v = \perp_v[\mathbf{this}/nc, v_1/t_1, ..., v_n/t_n, w_1/tt_1, ..., w_m/tt_m] \\ ((\epsilon, \epsilon_v), Expr) \vdash \tau_1 e_1, ..., \tau_l e_l \\ (\tau_1 ... \tau_l) \in dom(\epsilon_c(\pi(nc))) \\ ((\epsilon, \epsilon_v), Instr, \mathbf{void}) \vdash i_1, ..., i_{k+1} \\ i_{k+1} = \mathbf{return}\end{array}}{(\epsilon, Declconstr, nc) \vdash nc\ (t_1 v_1 ... t_n v_n)\ (w_1 tt_1 ... w_m tt_m)\ \mathbf{super}\ (\tau_1 e_1 ... \tau_l e_l)\ (i_1 ... i_{k+1})}$$

1.3.3.5 Déclaration de champ bien typée

Pour qu'une déclaration de champ soit bien typée, il suffit que l'expression correspondant à la valeur initiale du champ soit bien typée et que son type soit une spécialisation du type déclaré pour le champ.

$$\frac{((\epsilon, \bot_v[\mathbf{this}/nc]), Expr) \vdash t'\ expr \quad t' \preceq_\pi t}{(\epsilon, Declchamp, nc) \vdash t\ nch\ (t'\ expr)}$$

1.3.3.6 Définition de classe bien typée

La vérification du "typage" d'une définition de classe se réalise uniquement relativement à un environnement global.

On s'attend à ce qu'une définition de classe soit bien typée si toutes les déclarations qui la composent sont bien typées (relativement au nom de la classe). Il faut cependant ajouter qu'on doit au moins disposer dans la classe d'un "premier constructeur".

$$\frac{(\epsilon, Declchamp, nc) \vdash dch_1, ..., dch_n \quad (\epsilon, Declmethode, nc) \vdash dm_1, ..., dm_k \quad (\epsilon, Declconstr, nc) \vdash dc_1, ..., dc_m \quad \exists i : 1 \leq i \leq m : dc_i = nc\ (t_1 v_1 ... t_l v_l)\ (tt_1 w_1 ... tt_p w_p)\ \mathbf{prem}\ instr^+}{(\epsilon, Defclass) \vdash nc\ (dch_1 ... dch_n)\ (dm_1 ... dm_k)\ (dc_1 ... dc_m)}$$

$$\frac{(\epsilon, Declchamp, nc) \vdash dch_1, ..., dch_n \quad (\epsilon, Declmethode, nc) \vdash dm_1, ..., dm_k \quad (\epsilon, Declconstr, nc) \vdash dc_1, ..., dc_m \quad \exists i : 1 \leq i \leq m : dc_i = nc\ (t_1 v_1 ... t_l v_l)\ (tt_1 w_1 ... tt_p w_p)\ \mathbf{super}\ expr^*\ instr^+}{(\epsilon, Defclass) \vdash nc\ (\mathbf{extend}\ np)\ (dch_1 ... dch_n)\ (dm_1 ... dm_k)\ (dc_1 ... dc_m)}$$

1.3.3.7 Programme bien typé

Cette dernière définition doit tout simplement exprimer l'adéquation de l'environnement global et du programme.

Nous abandonnons ici la formalisation par "règles" qui s'avèrerait trop lourde au niveau des notations.

Un programme *prog* est bien typé s'il vérifie les trois conditions ci-après.

1. On peut construire un environnement global primitif $\epsilon = (\epsilon^0_{ch}, \epsilon^0_m, \epsilon^0_c, \pi)$ tel que pour toute

définition de classe $defclass$ de $prog$ de la forme

$$nc\ [\textbf{extend}\ np]\ (dch_1...dch_n)\ (dm_1...dm_k)\ (dc_1...dc_m)$$

on ait les propriétés suivantes.

(a) Si la partie [**extend** np] n'est pas présente alors nc n'appartient pas au domaine de π, sinon $\pi(nc) = np$.

(b) Si chaque déclaration dch_i de champs se décompose en $(t_i\ nch_i\ expr_i)$, alors

$$dom(\epsilon^0_{ch}\ nc) = \{nch_1, ..., nch_n\} \wedge (\epsilon^0_{ch}\ nc\ nch_i = t_i).$$

(c) Si chaque déclaration de méthode d_m se décompose en

$$tr_i\ methode_i\ (t_i^1 v_i^1 ... t_i^{n_i} v_i^{n_i})\ [corps_i]$$

alors

$$dom(\epsilon^0_m\ nc\ methode)$$
$$= \{(tt_1...tt_l) \mid \exists i : 1 \leq i \leq k : methode = methode_i \wedge (tt_1...tt_l) = (t_i^1...t_i^{n_i})\},$$

$$\forall i : 1 \leq i \leq k : \epsilon^0_m\ nc\ methode_i\ (t_i^1...t_i^{n_i}) = tr_i.$$

(d) Si chaque déclaration de constructeur dc_i se décompose en

$$nc\ (t_i^1 v_i^1 ... t_i^{n_i} v_i^{n_i})\ corps_i$$

alors

$$dom(\epsilon^0_c\ nc) = \{(t_1^1...t_1^{n_1}), ..., (t_m^1...t_m^{n_m})\}.$$

2. Toutes les définitions de classes sont bien typées relativement à ϵ.

3. Il existe une et une seule classe comportant une déclaration de méthode de signature $main()$.

1.4 Syntaxe abstraite et points de programme

Nous introduisons maintenant une troisième et dernière syntaxe, nommée **SAP**, qui sera en fait la seule syntaxe utilisée dans les chapitres suivants.

1.4.1 Syntaxe abstraite "labelisée" (SAP)

La syntaxe **SAP** se construit à partir de **SAT** : la modification principale apportée est l'introduction de "labels" d'une part au niveau des instructions et d'autre part au niveau des déclarations de méthodes et de constructeurs.

Les "labels" doivent identifier des points de programme (i.e. ils doivent permettre de se situer de manière univoque dans un programme donné). Il est par conséquent évident que ces "labels"

ne peuvent être quelconques : la définition des propriétés souhaitables pour ceux-ci sera l'objet de la section suivante.

Ce besoin d'identifier des points de programme découle du choix effectué au niveau de la sémantique : en effet, nous avons opté pour une sémantique opérationnelle sous forme de "système de transitions". Remarquons au passage que, si nous avions permis une évaluation des expressions avec effet de bord, nous aurions également dû ajouter des "labels" au niveau des expressions.

Pour coller plus encore à cette vue orientée "organigramme" des programmes, nous transformons aussi quelque peu les instructions au niveau du **if** et du **while**.

Signalons enfin que dorénavant nous supposerons que tous les programmes manipulés sont "bien typés" au sens défini dans la section précédente.

La définition de la syntaxe **SAP** est fournie par la figure 1.3.

1.4.2 Syntaxe abstraite bien "labelisée"

Précisons maintenant les propriétés que doivent vérifier les labels d'un programme de **SAP** pour permettre de décrire le "déroulement d'une exécution".

1.4.2.1 Séquence d'instructions bien labelisée

Intuitivement, il faut que dans le corps d'une méthode, chaque label "initial" permette d'identifier de manière univoque l'instruction auquel il est attaché; d'autre part, d'une instruction quelconque (mis à part une instruction **return**), on doit "passer" à une autre instruction du corps de la méthode.

Formulons maintenant ces idées de manière plus précise.

Point de vue "labels", on peut distinguer les trois "patterns" d'intructions suivants.

1. Le cas général consiste en une "instruction" encadrée par un label initial et un label final : *lab instr lab*.
2. Une instruction de test possède un label initial et deux labels finaux : *lab* **if** *expr lab lab*.
3. Une instruction **return** possède uniquement un label initial : *lab* **return**.

Ces trois "patterns" peuvent être couverts par la forme générale : $(d, i, \mathbf{f})$ avec $d \in Lab$, $\mathbf{f} = (f^1, ..., f^n) \in Lab^n$ et $n \in \{0, 1, 2\}$.

Ensembles atomiques

lab
litt
nclasse, *nvar*, *nmethode*, , *nchamp*

Ensembles construits

prog	::=	$defclass^+$
defclass	::=	*nclasse* [**extend** *nclasse*] $declchamp^*$ $declmethode^*$ $declconstr^+$
type	::=	**bot** | **int** | **bool** | *nclasse* | **void**
declchamp	::=	*type nchamp expr*
declmethode	::=	*type nmethode* $(type\ nvar)^*$ | *type nmethode* $(type\ nvar)^*$ $(type\ nvar)^*$ *lab* $instr^+$
declconstr	::=	*nclasse* $(type\ nvar)^*$ $(type\ nvar)^*$ *lab* **prem** *lab* $instr^+$ | *nclasse* $(type\ nvar)^*$ $(type\ nvar)^*$ *lab* **super** $expr^*$ *lab* $instr^+$ | *nclasse* $(type\ nvar)^*$ $(type\ nvar)^*$ *lab* **this** $expr^*$ *lab* $instr^+$
instr	::=	*lab* **affect** *des expr lab* | *lab* **if** *expr lab lab* | *lab* **skip** *lab* | *lab* **proc** *appel lab* | *lab* **return** [*expr*] | *lab* **fonc** *nvar appel lab* | *lab* **constr** *nvar nclasse* $expr^*$ *lab*
appel	::=	*type* **this** *nmethode* $expr^*$ | *type* **super** *nmethode* $expr^*$ | *type nvar nmethode* $expr^*$
des	::=	*type nvar* | *type nchamp* | *type desinst nchamp*
desinst	::=	*type* **this** | *type* **super** | *des*
expr	::=	*type* **null** | *type litt* | *type op* $expr^*$ | *desinst*

Figure 1.3: Définition de **SAP**

La séquence d'instructions

$$((d_1, i_1, \mathbf{f}_1), ..., (d_m, i_m, \mathbf{f}_m))$$

est dite "bien labelisée" si et seulement si

1. les labels d'une même instruction sont tous distincts,
2. $\forall i, j : 1 \leq i, j, \leq m : i \neq j \Rightarrow d_i \neq d_j$,
3. $\forall i, j : (1 \leq i \leq m) \wedge (1 \leq j \leq n_i) : (\exists k : 1 \leq k \leq m : d_k = f_i^j)$.

1.4.2.2 Déclaration de méthode bien labelisée

La déclaration de méthode concrète (où *param* et *varloc* appartiennent à $(Type\ Nvar)^*$)

$$t\ methode\ param\ varloc\ lab\ ((d_1, i_1, \mathbf{f}_1)...(d_m, i_m, \mathbf{f}_m))$$

est dite "bien labélisée" si et seulement si

1. $lab = d_1$,
2. $((d_1, i_1, \mathbf{f}_1), ..., (d_m, i_m, \mathbf{f}_m))$ est bien labelisée.

1.4.2.3 Déclaration de constructeur bien labelisée

La déclaration de constructeur

$$nc\ param\ varloc\ l_0\ \mathbf{prem}\ l_1\ ((d_1, i_1, \mathbf{f}_1)...(d_m, i_m, \mathbf{f}_m))$$

est dite "bien labelisée" si et seulement si

1. $l_1 = d_1$,
2. $((d_1, i_1, \mathbf{f}_1), ..., (d_m, i_m, \mathbf{f}_m))$ est bien labelisée,
3. $\forall i : 1 \leq i \leq m : l_0 \neq d_i$.

La définition est évidemment identique pour les deux autres formes de constructeur.

1.4.2.4 Programme bien labelisé

Un programme est bien labelisé si et seulement si

1. les ensembles de labels des différentes déclarations de méthodes et déclarations de constructeurs intervenant dans le programme sont disjoints deux à deux,
2. ces déclarations sont bien labelisées.

1.4.3 Exemple de "labelisation"

Nous donnons, dans cette section, un exemple de procédure permettant de "labeliser" un programme de **SAT** (i.e. permettant de transformer un programme de **SAT** en un programme de **SAP** bien labelisé).

Pour différencier les ensembles constituants de **SAT** des ensembles constituants de **SAP**, on apposera un "p" en indice à ces derniers.

Nous choisissons arbitrairement de considérer Lab comme un sous-ensemble de $\mathbb{N}$.

1.4.3.1 Labelisation d'une instruction

Dans un premier temps, expliquons comment transformer une instruction de **SAT** en une séquence d'instructions de **SAP**. Ce renvoi de séquences d'instructions s'explique par la double "tâche" de la procédure de "labelisation" : cette procédure doit, non seulement, ettiqueter correctement les instructions mais doit aussi transformer les structures de test et de boucle.

Nous définissons simultanément les deux fonctions suivantes :

$$\tau : Instr \longrightarrow Lab \longrightarrow (Instr_p)^* \times Lab$$
$$\tau^* : Instr^* \longrightarrow Lab \longrightarrow (Instr_p)^* \times Lab$$

Si $\tau\ instr\ l = (liste, l')$, $liste$ correspond à une séquence d'instructions bien labelisée "équivalente" à $instr$ dont le "label" initial est l. Le label l' correspond quant à lui au label suivant "utilisable" ("suivant" est ici à prendre dans le sens de l'ordre usuel sur $\mathbb{N}$).

La fonction τ^* applique simplement la fonction τ "en cascade" à une liste d'instructions.

Dans ce qui suit, nous employons les crochets pour délimiter les listes et le symbole "+" pour

représenter la concaténation.

$\tau(\textbf{affect}\ des\ expr)\ l = ([l\ \textbf{affect}\ des\ expr\ (l+1)], l+1)$
$\tau(\textbf{proc}\ appel)\ l = ([l\ \textbf{proc}\ appel\ (l+1)], l+1)$
$\tau(\textbf{fonc}\ nvar\ appel)\ l = ([l\ \textbf{fonc}\ nvar\ appel\ (l+1)], l+1)$
$\tau(\textbf{constr}\ nvar\ nclasse\ expr^*)\ l = ([l\ \textbf{constr}\ nvar\ nclasse\ expr^*\ (l+1)], l+1)$
$\tau(\textbf{return}\ expr)\ l = ([l\ \textbf{return}\ expr], l+1)$
$\tau(\textbf{while}\ expr\ instr^*)\ l = ([l\ \textbf{if}\ expr\ (l+1)\ fin+1] + liste + [fin\ \textbf{skip}\ l], fin+1)$
où $(liste, fin) = \tau^*\ instr^*\ (l+1)$
$\tau(\textbf{if}\ expr\ linst_1\ linst_2)\ l = ([l\ \textbf{if}\ expr\ (l+1)\ (f_1+1)] + liste_1 + [f_1\ \textbf{skip}\ f_2] + liste_2, f_2)$
où $\begin{cases} \tau^*\ linst_1\ (l+1) = (liste_1, f_1) \\ \tau^*\ linst_2\ (f_1+1) = (liste_2, f_2) \end{cases}$

$\tau^*([])\ l = ([], l)$
$\tau^*([inst] + linst)\ l = (liste + p_1(\tau^*(liste)\ fin), p_2(\tau^*(liste)\ fin))$
où $(liste, fin) = \tau(inst)\ l$

1.4.3.2 Labelisation d'une déclaration

Pour "labeliser" une déclaration de méthode concrète à partir d'un label donné l, il suffit d'attribuer le label l à la méthode et d'appliquer la fonction τ^* au corps de la méthode (le second argument de τ^* étant bien sûr l).

Pour "labeliser" une déclaration de constructeur à partir d'un label donné l, on attribue le label l au constructeur; le label "intermédiaire" est alors $l+1$ et on applique la fonction τ^* au corps du constructeur (le second argument de τ^* étant $(l+1)$).

1.4.3.3 Labelisation d'un programme

Pour "labeliser" un programme, on étiquette séquentiellement les déclarations de méthodes et de constructeurs du programme : par exemple, on "labelise" la première déclaration à partir de zéro; la fonction τ^* renvoie le "label libre suivant" l; on "labelise" la seconde déclaration à partir de l et ainsi de suite jusqu'à la fin du programme.

1.4.4 Quelques définitions supplémentaires

Pour clôturer ce chapitre, définissons quelques fonctions syntaxiques qui s'avéreront utiles lors de la rédaction de la sémantique. Ces fonctions sont correctement définies pour peu que l'on démarre d'un programme de **SAP** bien typé et bien labelisé[6].

[6] On devrait plutôt écrire "de la transformation bien labelisée d'un programme de **SAT** bien typé".

On se contente pour définir ces fonctions de préciser leur signature et de décrire informellement leur signification.

Définition 1.15 (Champs d'une classe)

Etant donné un programme bien typé, la fonction Ch renvoie toutes les informations relatives aux champs fournie dans la déclaration d'une classe donnée.

$Ch : Nclasse \nrightarrow (Type\ Nchamp\ Expr)^*$

Définition 1.16 (Location initiale d'une méthode)

Etant donné un programme bien typé et bien labelisé, la fonction λ_m détermine à partir d'un nom de classe, d'un nom de méthode et d'une liste de types la déclaration de méthode adéquate (si celle-ci existe).

$\lambda_m : Nclasse \longrightarrow Nmethode \longrightarrow (Type)^* \nrightarrow Infomethode$

où

$Infomethode = (Lab\ (Type\ Nar)^*\ (Type\ Nvar)^*) + \{\textbf{abstract}\}$

Cette fonction se définit en fait en suivant un chemin tout à fait analogue à celui emprunté lors de la définition de la fonction ϵ_m (cf. définitions 1.7, 1.8 et 1.9). Cette fonction renvoie juste un un autre genre d'information.

Définition 1.17 (Location initiale d'un constructeur)

Etant donné un programme bien typé et bien labelisé, la fonction λ_c détermine à partir d'un nom de classe et d'une liste de types la déclaration de constructeur adéquate (si celle-ci existe).

$\lambda_c : Nclasse \longrightarrow (Type)^* \nrightarrow (Lab\ (Type\ Nar)^*)$

Cette fonction se définit, quant à elle, de manière similaire à le fonction ϵ_c (cf. définitions 1.10 et 1.11).

Chapitre 2

Sémantique opérationnelle

Ce chapitre constitue "l'aboutissement" de la première partie de ce travail puisqu'il présente une sémantique opérationnelle du langage défini au cours du chapitre précédent. Rappelons à ce sujet qu'à partir de maintenant, nous ne manipulons plus que des programmes respectant la syntaxe **SAP** et que nous supposons correctement *typés* et *labelisés*.

Comme nous l'avons déjà signalé, la sémantique présentée ici est une sémantique opérationnelle traduite sous forme de système de transitions. Une autre caractéristique majeure de la sémantique présentée dans ce chapitre est sans doute, au-delà de notations parfois un peu "fastidieuses", son aspect naturel. En effet, l'allure de celle-ci rappelle directement les descriptions intuitives d'une exécution de programme.

Par cette dernière caractéristique, cette sémantique s'oppose à la sémantique exposée dans le chapitre suivant qui a plutôt pour but de permettre une analyse, par interprétation abstraite, de difficulté acceptable.

Ce chapitre se divise en deux parties. La première se consacre exclusivement à l'explication de ce que nous appelons le *domaine d'interprétation*. Cette notion recouvre en fait les concepts classiques d'environnement et de store; c'est en effet via ce type d'objets qu'on peut donner une *valeur* (i.e. interpréter) à des composants syntaxiques tels que les expressions et les désignateurs.

La seconde partie s'attaque quant à elle au système de transitions lui-même. La majorité de cette section est donc dédiée à l'énoncé des règles de celui-ci. On débute cependant la dite section par un bref rappel concernant l'interprétation à donner au système présenté.

2.1 Domaine d'interprétation

Dans cette section, on commence par préciser ce qu'on entend exactement par *valeur*. A partir de là, on peut définir notre domaine d'interprétation et définir l'évaluation d'un désignateur et d'une expression.

Remarquons dès à présent que, vu l'absence d'appels de fonction et de constructeur dans les expressions, ces évaluations ne modifient en rien le domaine d'interprétation et peuvent se définir facilement selon une optique plus "dénotationnelle".

Soulignons une propriété que nous aurions pu mentionner au préalable : toutes les fonctions partielles manipulées dans ce travail sont de domaine fini.

2.1.1 Valeurs

On distingue deux catégories de valeurs : les valeurs "de base" et les instances. Les valeurs de base correspondent d'une part aux littéraux (entiers et booléens) et d'autre part aux objets des différents types "classes" sans structure (i.e. non initialisés ou mis à **null**). Les instances correspondent aux éléments "de type classe" "existant réellement" .

Chaque valeur contient un type[1] que nous désignerons par la locution "type dynamique".

Définition 2.1 (Ensembles de valeurs)

$\mathbb{B}ase$ désigne l'ensemble des valeurs de base.

$$\mathbb{B}ase = \{\mathbf{bool}\} \times \mathbb{B}ool + \{\mathbf{int}\} \times \mathbb{Z} + Type \times \{\mathbf{ni}\}$$

$\mathbb{I}nst$ désigne l'ensemble des instances.

$$\mathbb{I}nst \subseteq Nclasse \times (Nchamps + \{\mathbf{super}\} \rightarrowtail \mathbb{L}oc)$$

Si (nc, v) appartient à $\mathbb{I}nst$, alors v est injective.

$$\mathbb{V}al = \mathbb{B}ase + \mathbb{I}nst$$

$\mathbb{L}oc$ désigne ce que nous appellerons l'ensemble des locations. On peut, grossièrement, voir les locations comme des adresses; nous reviendrons plus précisément sur leur signification intuitive par la suite.

Il semble que l'on modélise ici deux choses différentes de la même manière : "x vaut **ni**" peut signifier "x est non initialisé" mais aussi "on a affecté **null** à x".

La distinction est en fait réalisée au niveau du type. Dans le cas d'une affectation à **null**, le type dynamique est **bot**; ce qui ne sera jamais le cas pour une "variable" non initialisée.

[1] Il est évident que le type fictif **void** n'a aucun sens comme composant de la "valeur" d'une variable. Dorénavant, l'ensemble $Type$ correspond à l'ensemble $Nclasse + \{\mathbf{bot}, \mathbf{bool}, \mathbf{int}\}$.

2.1.2 Environnement et store

Ces deux notions ont ici une allure assez classique : l'environnement fournit la location associée à chaque variable ainsi qu'au désignateur **this**. Il fournit également le type statique des ces éléments. Le store associe une valeur aux différentes locations.

On dira souvent que l'environnement fournit les adresses des éléments "directement accessibles".

Définition 2.2 (Environnements et stores)

$\mathbb{E}nv$ désigne l'ensemble des environnements.

$$\mathbb{E}nv \subseteq Nvar + \{\mathbf{this}\} \rightarrowtail Type \times \mathbb{L}oc$$

Si e appartient à $\mathbb{E}nv$ et si x et y sont des éléments distincts du domaine de e, alors

$$p_2(e(x)) \neq p_2(e(y)).$$

$\$tore$ désigne l'ensemble des stores.

$$\$tore = \mathbb{L}oc \rightarrowtail \mathbb{V}al$$

Dans la définition ci-avant, $p_2(e(x))$ désigne la deuxième composante de $e(x)$. Plus généralement si $A_1, ..., A_n$ sont des ensembles, on notera p_i le i-ème projecteur défini par

$$\begin{array}{rccc} p_i : & A_1 \times ... \times A_n & \longrightarrow & A \\ & (a_1, ..., a_n) & \rightsquigarrow & a_i. \end{array}$$

La propriété d'injectivité exigée pour la deuxième projection de l'environnement traduit l'absence "d'aliasing" au niveau des noms de variables résultant du fait qu'il n'y a pas de passage par adresse.

On note $\perp_0$ l'élément de $\mathbb{E}nv$ de domaine vide et $\perp_1$ l'élément de $\$tore$ de domaine vide.

Ces deux constituants, environnement et store, forment évidemment la structure de notre domaine d'interprétation mais il est clair que tous les couples environnement/store ne représentent pas des situations "réalistes". La définition du domaine réside maintenant dans l'énoncé de propriétés restreignant le nombre de couples de l'espace $\mathbb{E}nv \times \$tore$.

Par la suite, on représentera souvent, dans les exemples, les couples environnement/store par des graphiques. Ainsi le graphique de la figure 2.1 symbolise une situation où l'environnement permet d'accéder à 2 variables x et y. Il permet évidemment également d'accéder à l'instance courante. A la variable x correspond la valeur entière 7; à la variable y correspond une instance de type A dont l'unique champ *ch* est non initialisé; l'instance courante est, quant à elle, de type C et son instance "père" de type D (cette dernière ne possède qu'un seul champ *ch* booléen à **true**). Les entiers inscrits "à l'origine des flèches" symbolisent les locations. Remarquons que cette représentation graphique ne reprend pas toute l'information d'un couple environnement/store puisque que, nulle part, elle ne mentionne les types statiques des variables.

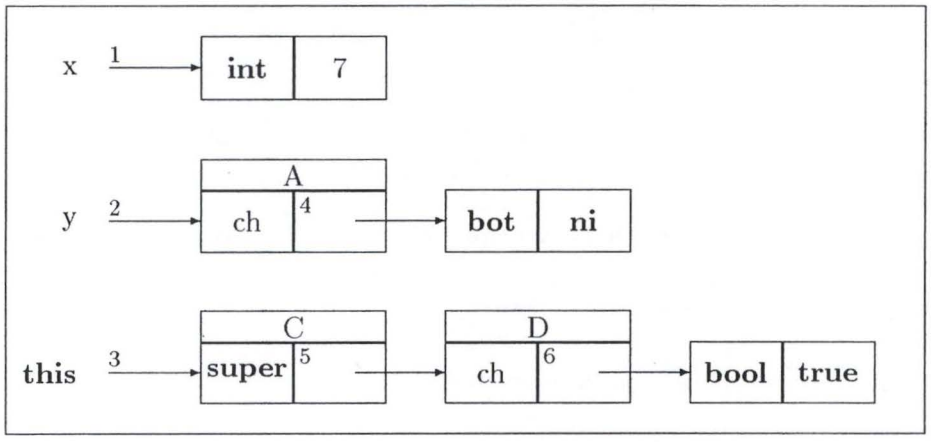

Figure 2.1: Un exemple d'élément du domaine

2.1.3 Définition du domaine

Enonçons maintenant les différentes propriétés intuitivement souhaitables du domaine. Soit donc (d_0, d_1) un élément du produit cartésien $\mathbb{E}nv \times \mathbb{S}tore$, que doit vérifier ce couple pour être acceptable ?

Propriété 1 : forme de l'environnement

L'environnement doit permettre d'accéder à l'instance courante (i.e. **this** doit appartenir à son domaine). Aucun type statique ne peut valoir **bot** (type réservé pour l'expression **null**) et le type statique de l'instance courante ne peut être qu'un type classe.

L'environnement doit donc être de la forme

$$d_0 = \bot_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \mathbf{this}/t_{n+1}\delta_{n+1}]$$

avec

$$n \in \mathbb{N},\ t_1, ..., t_n \neq \mathbf{bot},\ t_{n+1} \in Nclasse.$$

Dans la formule précédente $t_i\delta_i$ dénote, de manière abrégée, le couple (t_i, δ_i).

Propriété 2 : unicité des locations

Une des caractéristiques principales de notre interprétation du domaine se traduit dans l'exigence que toutes les locations manipulées doivent être "uniques". Dans cet ordre d'idées, on a déjà imposé l'injectivité des fonctions intervenant dans les instances et l'injectivité de l'environnement au niveau des locations; on ajoute maintenant la condition suivante.

$$d_1(l) = (nc, v) \wedge d_1(l') = (nc', v') \Rightarrow (d_1(l) = d_1(l') \vee Im(v) \cap Im(v') = \emptyset)$$

Propriété 3 : cohérence du domaine

Toutes les locations accessibles directement par l'environnement ou indirectement (i.e. en passant par le store) doivent appartenir au domaine du store, i.e. si on définit la suite de niveaux

$$D_0 = \{\delta_i \mid 1 \leq i \leq n+1\}$$

$$D_{j+1} = D_j \cup \bigcup_{\substack{l \in D_j \\ d_1(l) \in \mathbb{I}nst}} Im(p_2(d_1(l)))$$

qui est évidemment croissante et converge vers l'ensemble de locations D, le domaine du store doit respecter l'inclusion $D \subseteq dom(d_1)$.

Il se peut cependant que certaines locations ne soient pas accessibles, directement ou indirectement, via l'environnement. Si des instances sont associées à de telles locations, les locations mentionnées dans ces ensembles doivent quand même appartenir au domaine du store, i.e.

$$(l \in dom(d_1) \wedge d_1(l) = (nc, v) \in \mathbb{I}nst) \Rightarrow Im(v) \subseteq dom(d_1).$$

Propriété 4 : respect des types statiques

Cette propriété s'exprime relativement à un environnement global de types noté ϵ.

Il faut que la structure des instances dans le store respecte les définitions de classes du programme; c'est à dire que si $d_1(l) = (nc, v)$ est un élément de $\mathbb{I}nst$, le store doit vérifier les conditions ci-après.

- **super** $\in dom(v) \Leftrightarrow nc \in dom(\pi)$
- $\forall ch \in Nchamp, ch \in dom(v) \Leftrightarrow ch \in dom(\epsilon^0_{ch} nc)$
- **super** $\in dom(v) \Rightarrow d_1(v(\textbf{super}))) = (\pi(nc), p) \in \mathbb{I}nst$
- $ch \in dom(v) \Rightarrow p_1(d_1(v(ch))) \preceq_\pi (\epsilon^0_{ch}\ nc\ ch)$

Le type dynamique d'un champ peut être une spécialisation du type statique de celui-ci et sa valeur peut être indéfinie (dans le sens "mis à **null**"); il n'en va naturellement pas de même pour le type de **super** qui doit être exactement celui fourni par la relation d'extension; la valeur de ce pseudo-champ doit également être une instance (toute instance "contient" l'instance de son "père").

Il faut également que les types dynamiques des valeurs directement accessibles par l'environnement soient cohérents avec les types statiques de ce dernier (qui correspondraient en fait à ceux d'un environnement de types pour les variables), i.e.

$$\forall i : 1 \leq i \leq n : p_1(d_1(\delta_i)) \preceq_\pi t_i,$$

$$p_1(d_1(\delta_{n+1})) = t_{n+1}.$$

Soulignons, une fois encore, que cette propriété n'a de sens que vis à vis de ϵ c'est-à-dire vis à vis d'un programme donné. Le domaine d'interprétation que nous définissons actuellement est donc propre à un programme donné.

Définition 2.3 (Domaine d'interprétation d'un programme)

Soit p un programme et ϵ son environnement global de types. $\mathbb{D}_p$ désigne le domaine d'interprétation de p. Il s'agit du plus grand ensemble inclus dans le produit cartésien $\mathbb{E}nv \times \mathbb{S}tore$ tel que chacun de ses éléments d vérifie les propriétés 1,2,3 et 4.

Par la suite, on omettra de rappeler la dépendance du domaine à un programme donné (en supposant que l'on parle toujours du même programme p engendrant l'environnement global de types ϵ).

Si on regarde la propriété 3, on peut dire que si d appartient à $\mathbb{D}$, d "respecte" la relation d'extension π. L'intérêt de cette remarque réside dans le fait que π induit par conséquent une autre relation, non plus sur les types, mais sur les locations.

Définition 2.4 (Relation d'extension sur les locations)

Soit d un élement de $\mathbb{D}$, on note $<_\pi$ la relation induite par π sur les locations de d. Cette relation, propre à chaque élément de $\mathbb{D}$, se définit par

$$l' <_\pi l \Leftrightarrow (d_1(l) = (nc, v) \in \mathbb{I}nst) \wedge v(\textbf{super}) = l'.$$

Il est évident que, comme π est bien fondée, $<_\pi$ est également bien fondée.

2.1.4 Evaluation d'un désignateur

Classiquement, de l'évaluation d'une expression "gauche" (appelée dans notre cas désignateur) doit résulter finalement une "adresse". Nous ne dérogeons pas ici à la règle et fournissons comme résultat d'évaluation une location.

Si le "calcul" de l'adresse d'une variable est trivial puisque le résultat est fourni directement par l'environnement, il n'en va pas de même pour les désignateurs faisant intervenir des noms de champ. En effet, d'une part il se peut que le champ ne soit pas directement un champ de la classe, correspondant au type du désignateur d'instance, mais le champ d'une classe dont elle hérite et, d'autre part, dans le cas d'une "spécialisation", il faut faire attention à retrouver le champ correspondant au type statique du désignateur (il est possible que la "spécialisation" possède un champ du même nom).

Illustrons ces considérations, un peu confuses, par un exemple.

La figure 2.2 met en présence une variable x déclarée de type B et une instance courante de type C. Au niveau de l'héritage, on a les relations suivantes : B étend A, C et D étendent B. Lorsqu'on évalue le désignateur ch, le champ ch n'apparaissant pas directement dans la

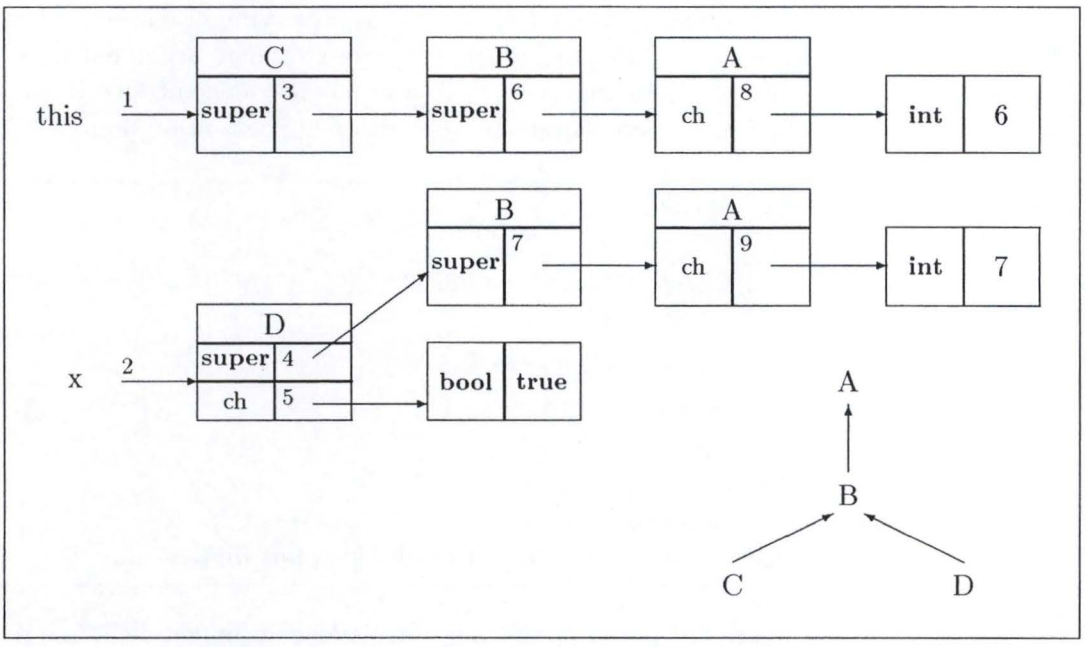

Figure 2.2: Evaluation d'un désignateur

classe C, il faut remonter le long de la "structure" de l'instance courante jusqu'à la classe A. Si on désire évaluer le désignateur $x.ch$, on ne peut pas se contenter d'appliquer le principe ci-avant sous peine de fournir comme résultat la location 5 qui n'est certainement pas "désignée statiquement" par $x.ch$ (dans le cas présent, elle n'est même pas associée à un type adéquat). Il faut, dans un premier temps, rechercher le long de l'instance correspondant à x la classe B et puis, seulement, appliquer le premier principe pour finalement aboutir à la location 9.

On formalise ce processus par l'intermédiaire de trois fonctions :

1. $\mathcal{A}drtype$ permet de retrouver l'instance exacte correspondant au type statique;
2. $\mathcal{A}drch$ permet de retrouver le champ recherché le long de la structure d'extension;
3. $\mathcal{A}dr$ utilise les deux fonctions précédentes pour calculer l'adresse d'un désignateur.

Définition 2.5 (Fonction $\mathcal{A}drtype$)

La fonction Adrtype renvoie l'adresse d'une valeur d'un type donné "à partir" d'une location donnée. Plus précisément, elle est définie par

$$\mathcal{A}drtype : Type \longrightarrow \$tore \longrightarrow \mathbb{L}oc \rightarrowtail \mathbb{L}oc$$

si $s(l) = (t, p)$
alors $\mathcal{A}drtype[\![t]\!]\ s\ l = l$
sinon si $s(l) = (t', v) \in \mathbb{I}nst \wedge t' \preceq_\pi t$
alors $\mathcal{A}drtype[\![t]\!]\ s\ l = \mathcal{A}drtype[\![t]\!]\ s\ v$(**super**)

La définition ci-dessus a bien un sens, du moins dans les cas où s symbolise le store d'un élement de $\mathbb{ID}$. En effet, cette définition se fait par induction sur le troisième argument de la fonction le long de la relation $<_\pi$ qui est alors bien fondée. Remarquons également que, dans ces cas, si $s(l) = (t', v) \in \mathbb{Inst} \wedge t' \preceq_\pi t$, on a nécessairement que **super** appartient au domaine de v.

Définition 2.6 (Fonction $\mathcal{A}drch$)

La fonction $\mathcal{A}drch$ renvoie l'adresse d'un champ donné "à partir" d'une location donnée. Plus précisément, elle est définie par

$$\mathcal{A}drch : Nchamp \longrightarrow \$tore \longrightarrow \mathbb{Loc} \rightarrowtail \mathbb{Loc}$$

si $s(l) = (nc, v) \in \mathbb{Inst}$
alors **si** $ch \in dom(v)$
alors $\mathcal{A}drch[\![ch]\!]\ s\ l = v(l)\ ch$
sinon **si** **super** $\in dom(v)$
alors $\mathcal{A}drch[\![ch]\!]\ s\ l = \mathcal{A}drch[\![ch]\!]\ s\ v(\textbf{super})$

De nouveau, cette définition se fait par induction sur le troisième argument de la fonction le long de la relation $<_\pi$.

On peut maintenant définir ce qu'est l'adresse d'un désignateur. Cette définition se construit, comme c'est l'usage, par induction sur la structure des désignateurs.

Définition 2.7 (Fonction $\mathcal{A}dr$)

Etant donné un élément de $\mathbb{ID}$, la fonction $\mathcal{A}dr$ fournit l'adresse d'un désignateur.

$$\mathcal{A}ch : Des \longrightarrow \mathbb{ID} \rightarrowtail \mathbb{Loc}$$

$$\mathcal{A}dr[\![t\ var]\!]\ (d_0, d_1) = p_2(d_0(var))$$
$$\mathcal{A}dr[\![t\ ch]\!]\ (d_0, d_1) = \mathcal{A}drch\ ch\ d_1\ p_2(d_0(\textbf{this}))$$
$$\mathcal{A}dr[\![t'\ (t\ \textbf{this})\ ch]\!]\ (d_0, d_1) = \mathcal{A}dr[\![t'\ ch]\!]\ (d_0, d_1)$$
$$\mathcal{A}dr[\![t'\ (t\ \textbf{super})\ ch]\!]\ (d_0, d_1) = \textbf{si}\ (d_1(p_2(d_0(\textbf{this}))) = (nc, v) \in \mathbb{Inst})\ \textbf{alors si}\ (\textbf{super} \in dom(v))\ \textbf{alors}\ \mathcal{A}drch[\![ch]\!]\ d_1\ v(\textbf{super})$$
$$\mathcal{A}dr[\![t'\ (t\ des)\ ch]\!]\ (d_0, d_1) = \textbf{si}\ (\mathcal{A}dr[\![t\ des]\!]\ (d_0, d_1) = l)\ \textbf{alors si}\ (\mathcal{A}drtype[\![t]\!]\ d_1\ l = ls)\ \textbf{alors}\ \mathcal{A}drch[\![ch]\!]\ d_1\ ls$$

2.1.5 Evaluation d'une expression

L'évaluation d'une expression fournit simplement une "valeur" au sens défini précédemment. La définition de la fonction d'évaluation $\mathcal{V}al$ se fait par induction sur la structure des expressions.

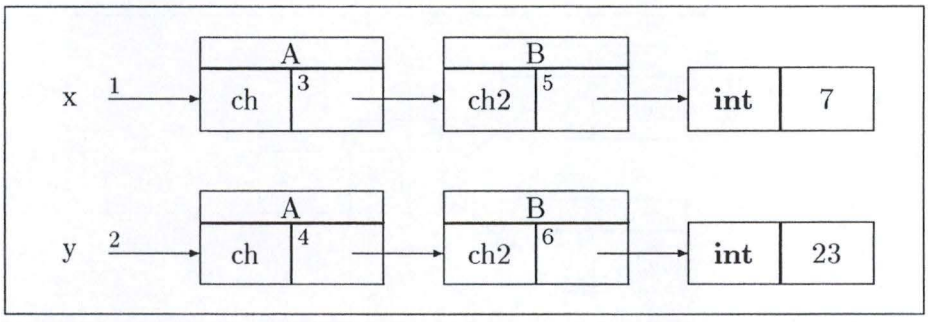

Figure 2.3: Situation simple sans "sharing"

Définition 2.8 (Fonction $\mathcal{V}al$)

Etant donné un élément de $\mathbb{D}$, la fonction $\mathcal{V}al$ fournit la valeur d'une expression.

$$\mathcal{V}al : Expr \longrightarrow \mathbb{D} \not\longrightarrow \backslash Val$$

$$\mathcal{V}al[\![t\ \mathbf{null}]\!]\,(d_0, d_1) = (\mathbf{bot}, \mathbf{ni})$$
$$\mathcal{V}al[\![t\ (op\ (e_1 ... e_n))]\!]\,(d_0, d_1) = \mathcal{V}_{op}(\mathcal{V}al[\![e_1]\!](d_0, d_1), ..., \mathcal{V}al[\![e_n]\!](d_0, d_1))$$
$$\mathcal{V}al[\![t\ litt]\!]\,(d_0, d_1) = [\![litt]\!]$$
$$\mathcal{V}[\![t\ \mathbf{this}]\!] = d_1(p_2(d_0(\mathbf{this}))$$
$$\mathcal{V}al[\![t\ \mathbf{super}]\!]\,(d_0, d_1) = \mathbf{si}\ d_1(p_2(d_0(\mathbf{this})) = (nc, v) \in \mathbb{I}nst \ \mathbf{alors}\ d_1(v(\mathbf{super}))$$
$$\mathcal{V}al[\![des]\!]\,(d_0, d_1) = \mathbf{si}\ (\mathcal{A}dr[\![des]\!]\,(d_0, d_1) = l) \ \mathbf{alors}\ d_1(l)$$

2.1.6 Remarques sur la signification du domaine

2.1.6.1 Représentation du "sharing"

Soit une situation où deux variables x et y de type A sont en présence[2]. Le type A possède un seul champ ch de type B qui lui-même possède un seul champ $ch2$ de type entier. La figure 2.3 représente une situation de ce genre où les structures de x et de y sont parfaitement distinctes.

La figure 2.4 schématise, quant à elle, une situation analogue mais qui indique, en outre, que les champs ch des deux objets x et y "partagent" la même instance de la classe B. Remarquons que le partage ne se traduit pas au niveau des locations mentionnées dans les instances x et y mais au niveau de l'image de celles-ci par le store.

Cette constation nous amène un léger problème. En effet, supposons maintenant que la classe B ne possède aucun champ, que penser de la situation symbolisée par la figure 2.5 ? S'agit-il

[2] Il faudrait normalement, pour chaque situation, représenter l'instance courante. Dans les différents exemples proposés, nous négligeons cette contrainte sauf si elle s'avère pertinente pour l'illustration en cours.

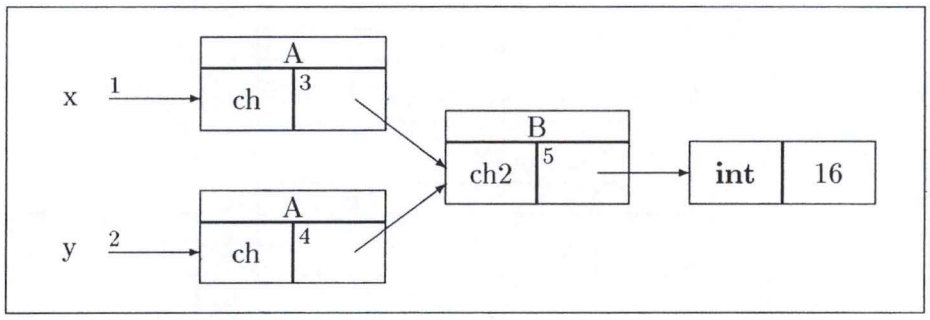

Figure 2.4: Situation simple avec "sharing"

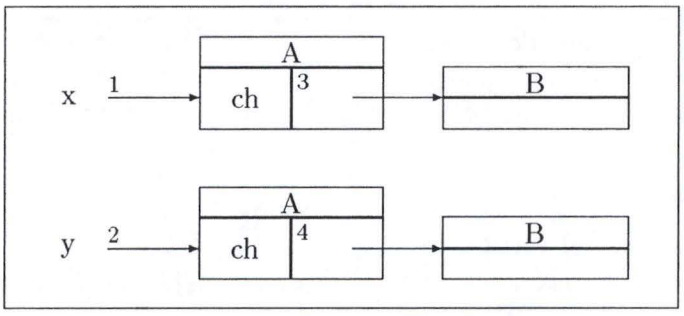

Figure 2.5: Situation avec ou sans "sharing" ?

d'une situation avec ou sans partage ? Le problème émerge du fait, que dans notre interprétation, l'identité des instances se définit au niveau des locations (uniques) attribuées à leurs champs. Par conséquent, si une classe ne possède pas de champ, rien ne permet dans notre modèle d'exprimer si deux instances de cette classe sont égales.

Pour lever ce problème, une solution s'impose : il suffit de supposer que toutes les classes possèdent au moins un champ. Au besoin, on ajoute un champ "factice" identificateur d'instance. La "valeur" de ce champ n'a pas d'importance en elle-même (on peut par exemple prendre des valeurs entières), ce qui compte est l'existence du champ et donc l'attribution d'une location à celui-ci en cas de création d'une instance.

2.1.6.2 Signification des "locations"

Avant d'aborder cette dernière remarque, précisons ce qui a motivé notre choix de domaine et notre définition des évaluations des désignateurs et des expressions. Notre objectif est de pouvoir modéliser l'effet d'une affectation par une simple modification ponctuelle de fonction : après évaluation de la location l correspondant au désignateur des et de la valeur v correspondant à l'expression $expr$, l'effet de l'affectation $des := expr$ se traduit simplement par la modification du store en l, la nouvelle image de l dans celui-ci étant v.

Revenons maintenant à notre dernière remarque. Pour éviter le problème de l'identité des instances, on aurait pu être tenté d'interpréter les locations différemment et abandonner ainsi

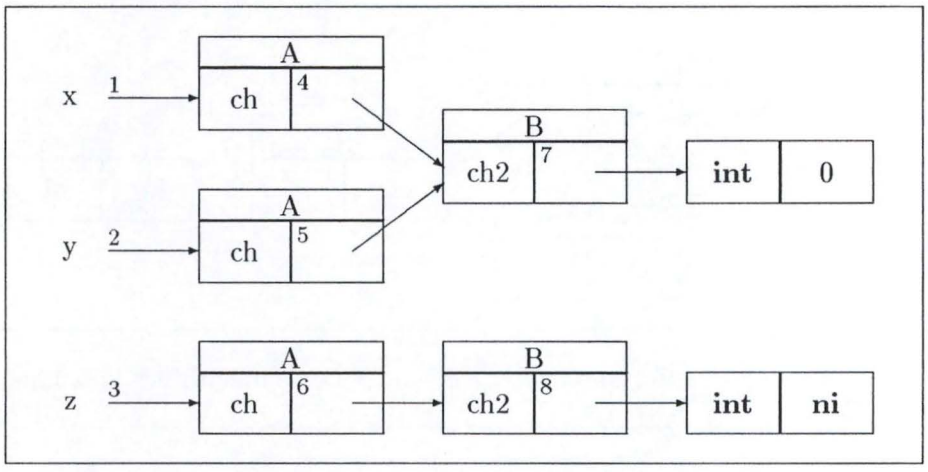

Figure 2.6: Situation S (avec unicité des locations)

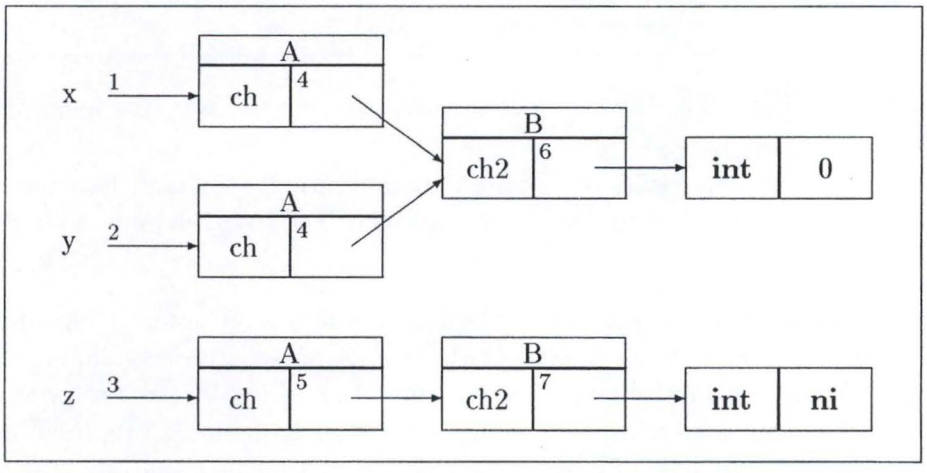

Figure 2.7: Situation S (sans unicité des locations)

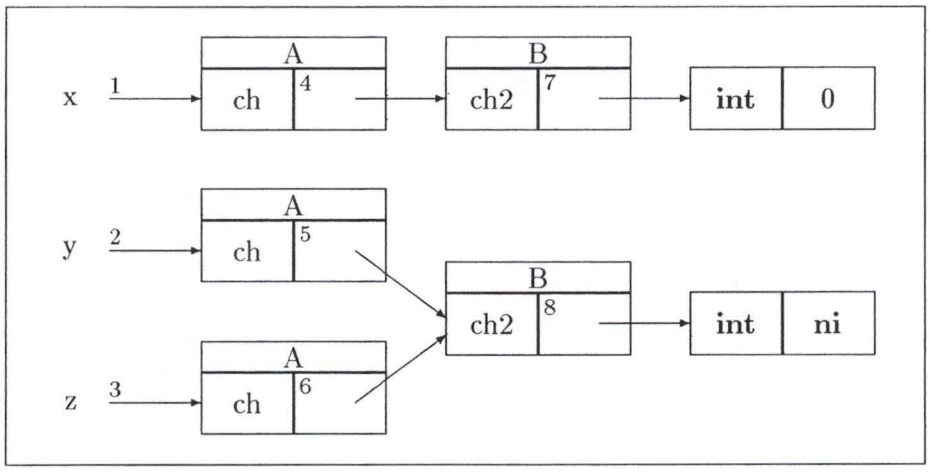

Figure 2.8: Effet de $y.ch := z.ch$ sur S (avec unicité des locations)

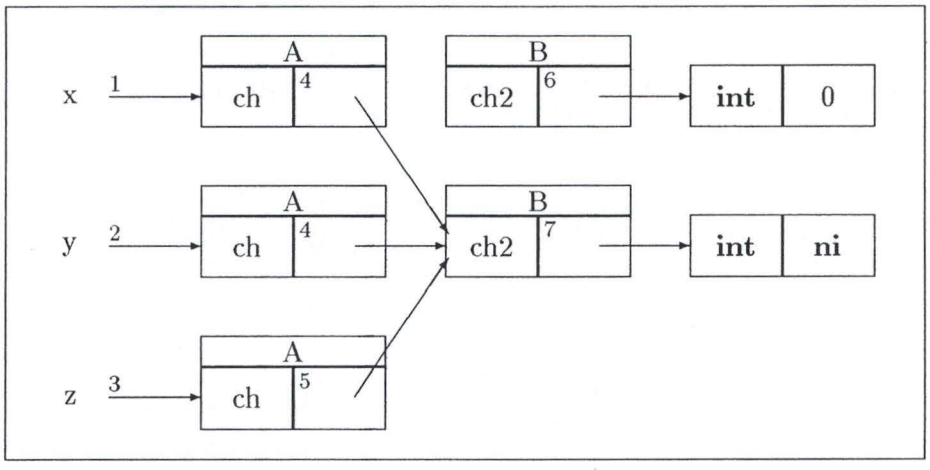

Figure 2.9: Effet de $y.ch := z.ch$ (sans unicité des locations)

la propriété d'unicité des locations : dans l'interprétation pour laquelle nous avons opté, les locations correspondent à des adresses de champs ou de variables, on peut, à l'inverse, interpréter les locations comme des adresses d'instance.

Représentons, dans chacune des deux optiques, une situation notée S où interviennent trois variables x, y et z, toutes trois de type A (le type A possède un seul champ noté ch de type B, possédant lui-même un seul champ entier noté $ch2$). Il existe, dans cette situation, un seul point de partage localisé au niveau du champ de x et y. La figure 2.6 représente cette situation dans notre domaine tandis que la figure 2.7 représente la même situation dans un domaine qui identifierait les instances via leur location.

Simulons maintenant l'exécution de l'instruction $y.ch := z.ch$ en employant la méthode évoquée au début de cette section. Dans notre domaine, nous obtenons la situation schématisée par la figure 2.8 tandis que la seconde interprétation nous conduit au résultat repris dans la figure 2.9.

Il est évident que ce dernier résultat ne correspond absolument pas à ce que l'on désire.

Les remarques présentées dans cette section ne signifient aucunement que le domaine qui identifie les instances via leurs adresses est non valide, elles signifient juste qu'il nécessite une autre modélisation de l'affectation que celle adoptée dans ce travail.

Une troisième solution eût été d'attribuer des locations à la fois aux champs et aux instances. Cette dernière solution a l'avantage de ne pas nécessiter de champ factice et de "coller" à notre modélisation de l'affectation. Nous l'avons cependant abandonnée car, vu la duplication des locations, elle engendre, d'une part, des complications de notations et, d'autre part, un traitement plus complexe au niveau abstrait.

2.2 Système de transitions

Cette section présente un système de transitions décrivant la sémantique opérationnelle d'un programme de **SAP** supposé bien typé et bien labelisé. La majeure partie de celle-ci est donc consacrée à l'expression des règles de transition du système. Celles-ci manipulent bien entendu des éléments du domaine présenté tout au long de la section précédente.

Commençons cependant par rappeler ce que nous entendons par système de transitions et quelle est l'intuition présente derrière celui-ci.

2.2.1 Présentation générale

Par système de transitions, nous entendons ici un triplet $T =< E, e_0, (\mathcal{R}_i)_{1 \leq i \leq k} >$ où

- E désigne un ensemble d'états,
- e_0 désigne un élement particulier de E appelé état initial,
- $(\mathcal{R}_i)$ une famille finie de règles appelées *règles de transition.*

Une règle de transition se présente sous la forme suivante.

Nom de règle

$$e \xrightarrow{Morceau\ de\ programme} e'$$

Conditions :

[*Liste de conditions*]

Dans ce format, e désigne l'état avant application de la règle *Nom de Règle* et e' l'état résultant de l'application de cette règle. *Liste de conditions* exprime, d'une part, des conditions que e doit vérifier pour que la règle puisse s'appliquer et, d'autre part, définit e' en fonction de e.

Morceau de programme exprime également une condition devant être vérifiée pour rendre la règle activable : e contient un label et *morceau de programme* débute par un label; pour que la règle soit applicable, il faut que ces deux labels coïncident.

Une règle de transition $\mathcal{R}_i$ s'interprète en fait comme une fonction partielle de l'ensemble des états dans lui-même : e appartient au domaine de $\mathcal{R}_i$ si et seulement si e vérifie les différentes conditions d'activation de $\mathcal{R}_i$ et, dans ce cas, $\mathcal{R}_i(e) = e'$.

Dans notre cas, le langage étant déterministe, il existe au plus une règle applicable par état (déterminée par le point de programme contenu dans cet état).

Une exécution du programme est décrite par une suite d'états (s_i), finie ou infinie, telle que

$$(s_0 = e_0) \wedge \forall i : i \geq 0 : (\exists j : 1 \leq j \leq k : s_{i+1} = \mathcal{R}_j(s_i)).$$

2.2.2 Ensemble des états

Définissons maintenant quel est, dans notre cas, l'ensemble des états.

On a déjà mentionné que chaque état devait contenir une location et il est évident que ceux-ci doivent manipuler des éléments de $\mathbb{ID}$.

De plus, il faut pouvoir décrire le mécanisme des appels de méthodes. Pour ce faire, la première idée qui vient à l'esprit est de disposer d'une pile où stocker les informations propres au contexte de chaque appel. C'est cette idée intuitive que nous décidons d'appliquer.

Définition 2.9 (Ensemble des états)

$\mathbb{E}tat$ dénote l'ensemble des états du système de transitions.

$$\mathbb{E}tat = Lab \times \mathbb{P}ile \times \mathbb{ID}$$
$$\mathbb{P}ile = \{(v_1, ..., v_n) \mid n \in \mathbb{N} \wedge \forall i : 1 \leq i \leq n : v_i \in \mathbb{E}nv \times Lab \times \mathbb{T}ypeappel\}$$
$$\mathbb{T}ypeappel = \{\mathbf{constr}\} \times (Nvar + \{\mathbf{this}, \mathbf{super}\}) + \{\mathbf{fonc}\} \times Nvar + \{\mathbf{proc}\}$$

Un état est donc constitué d'un label, d'un élément du domaine d'interprétation et d'une pile correspondant aux différents appels de méthodes et de constructeurs emboîtés. Sur la pile, on trouve les informations suivantes, toutes propres au contexte d'un appel de méthode ou de constructeur particulier : l'environnement au moment de l'appel, le point de programme correspondant à la fin de l'appel de méthode (appelé adresse de retour) et un indicateur du type d'appel; dans le cas d'un appel de constructeur ou de fonction, cette information est enrichie de "l'élément où renvoyer le résultat de l'appel".

La première projection de l'élément du domaine est appelée *environnement courant* par opposition aux différents environnements sauvegardés sur la pile. Le store est vu comme un "objet global" modifié tout au long de l'exécution.

On peut maintenant passer à la description des différentes règles de transition. On choisit, pour la présentation, de distinguer trois groupes de règles.

1. Les *règles simples* correspondent aux instructions non liées à des appels de méthodes, et coïncident grosso-modo aux règles qui n'ont pas d'effet sur la pile.
2. Les *règles relatives aux appels* s'ajustent avec des opérations de sauvegarde sur la pile, i.e. "d'empilage".
3. Inversément, les *règles relatives aux retours* correspondent, quant à elles, à des opérations de "dé-stockage" d'information, i.e à des opérations "de dépilage".

On ajoutera à ces règles une règle spéciale dite *de clôture* pour traduire la fin de l'exécution de la méthode initiale *main*.

2.2.3 Règles simples

L'effet de l'instruction vide est tout simplement de passer au point de programme suivant.

Skip

$$< l, P, d > \xrightarrow{l \ \mathbf{skip} \ l'} < l', P, d >$$

Conditions :

$$[\ /\ ]$$

De même, l'effet d'une instruction de test se traduit uniquement par un changement de position dans le programme. La nouvelle position est naturellement déterminée par le résultat de l'évaluation de l'expression du test.

If Vrai

$$< l, P, d > \xrightarrow{l \ \mathbf{if} \ expr \ l_1 \ l_2} < l_1, P, d >$$

Conditions :

$$\left[\ \mathcal{V}al[\![expr]\!]\, d = (\mathbf{bool}, true)\ \right]$$

If Faux

$$< l, P, d > \xrightarrow{l \ \mathbf{if} \ expr \ l_1 \ l_2} < l_2, P, d >$$

Conditions :

$$\left[\ \mathcal{V}al[\![expr]\!]\, d = (\mathbf{bool}, false)\ \right]$$

Comme on l'a déjà signalé, une affectation se traduit par une modification ponctuelle du store, à laquelle on adjoint évidemment un changement de point de programme.

Affect

$$< l, P, (d_0, d_1) > \xrightarrow{l \textbf{ affect } des\ expr\ l'} < l', P, (d_0, d_1[loc/val]) >$$

Conditions :

$$\left[\begin{array}{l} \mathcal{A}dr[\![des]\!]\,(d_0, d_1) = loc \\ \mathcal{V}al[\![expr]\!]\,(d_0, d_1) = val \end{array} \right]$$

2.2.4 Règles pour les appels

On distingue trois types d'appels : les appels de procédure, les appels de fonction et les appels de constructeur. Pour chacun de ces types d'appels, on stocke une information différente sur la pile.

2.2.4.1 Appel de procédure

Pour qu'un appel de méthode de la classe courante puisse avoir lieu, il faut évidemment que **this** désigne effectivement une instance. Si à partir du type, dynamique, de cette instance, du nom de la méthode et des types, statiques, des paramètres de l'appel, la fonction λ_m (cf. définition 1.16) parvient à déterminer une méthode du programme, on peut, à partir des informations renvoyées par cette fonction, construire le nouvel environnement de la méthode (cette fonction renvoie également le point de programme correspondant au début du corps de la méthode).

Le domaine de cet environnement comprend uniquement les paramètres formels et les variables locales de la méthode et, bien sûr, **this**. De nouvelles locations, i.e. n'appartenant pas au domaine du store, sont attribuées à tous ces éléments sauf à **this**. Au niveau du store, on attribue aux paramètres formels les valeurs des paramètres effectifs (évaluées "avant l'appel") et aux variables locales une valeur non initialisée. La valeur de **this** n'est naturellement pas modifiée.

On traduit en fait un classique "passage par valeur" pour les paramètres et un passage par adresse pour le paramètre "objet".

Soulignons encore que l'inexistence de variables globales "directes" se traduit par la restriction de l'environnement aux éléments de "l'en-tête" de la méthode.

Proc this

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \mathbf{proc}\ (tr\ \mathbf{this}\ meth\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, P + (d_0, lr, \mathbf{proc}), (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(\mathbf{this}))) = (nc, v) \in \mathbb{I}nst \\ \lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/d_0(\mathbf{this})] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ d'_1 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}] \end{array}\right]$$

Rappelons, qu'au niveau des types, on a les inégalités suivantes :

$$t_1 \preceq_\pi t'_1, ..., t_n \preceq_\pi t'_n,$$

$$p_1(\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1)) \preceq_\pi t_1, ..., p_1(\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1)) \preceq_\pi t_n.$$

Pour un appel de méthode relatif à une variable, la règle est tout à fait similaire. La seule différence se situe au niveau de l'image de **this** par le nouvel environnement qui doit ici correspondre à l'adresse de la variable. Soulignons que, dans le nouvel environnement, le type statique de **this** n'est pas l'ancien type statique de la variable sur laquelle se réalise l'appel mais bien son type dynamique.

Proc var

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \mathbf{proc}\ (tr\ var\ meth\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, P + (d_0, lr, \mathbf{proc}), (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(var))) = (nc, v) \in \mathbb{I}nst \\ \lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(nc, p_2(d_0(var)))] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ d'_1 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}] \end{array}\right]$$

Pour un appel relatif à l'instance "père" de la classe courante, la règle est de nouveau tout à fait analogue aux deux précédentes. Il faut, pour que l'appel puisse se dérouler, que **super** désigne bien une instance, ce qui est normalement le cas dans $\mathbb{I}D$ dès que **this** désigne une instance.

Proc super

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \mathbf{proc}\ (tr\ \mathbf{super}\ meth\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, P + (d_0, lr, \mathbf{proc}), (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(\mathbf{this}))) = (nc, v) \in \mathbb{I}nst \\ d_1(v(\mathbf{super})) = (np, u) \in \mathbb{I}nst \\ \lambda_m\ np\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(np, v(\mathbf{super}))] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ d'_1 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}] \end{array}\right]$$

2.2.4.2 Appel de fonction

Les règles traitant les appels de fonction sont presque identiques aux règles qui viennent d'être présentées pour les appels de procédure. On en dénombre, comme pour les procédures, trois : *Fonc this*, *Fonc super* et *Fonc var*.

Ces trois règles se déduisent chacune respectivement de leur homologue pour les procédures en modifiant juste l'information stockée sur la pile. En effet, il faut, dans le cas d'une fonction, savoir, au retour de l'appel, où "recopier" la valeur obtenue pour le résultat.

On ne présente par conséquent qu'une seule de ces trois règles, à titre d'illustration.

Fonc var

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \textbf{fonc}\ x\ (tr\ var\ meth\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, P + (d_0, lr, (\textbf{fonc}, x)), (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(var))) = (nc, v) \in \mathit{Inst} \\ \lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \textbf{this}/(nc, p_2(d_0(var)))] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ d'_1 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \textbf{ni}, ..., ll_m/tt_m\ \textbf{ni}] \end{array}\right]$$

2.2.4.3 Appel de constructeur

Les règles concernant les appels de constructeurs sont au nombre de quatre.

1. La règle *Constr var* correspond à l'appel de constructeur se situant au niveau des instructions.
2. La règle *Constr this* s'applique lors d'un appel à un constructeur de la forme **this**, elle traduit donc un appel à un autre constructeur de la classe.
3. La règle *Constr super* s'applique lors d'un appel à un constructeur de la forme **super**, elle traduit donc un appel à un constructeur de la classe "père".
4. La règle *Constr prem* est sans doute la plus particulière des quatre. Elle s'applique lorsque le constructeur est un constructeur ne faisant appel à aucun autre constructeur. Dans ce sens, elle ne correspond pas réellement à un appel et ne modifie absolument pas la pile des environnements (elle pourrait donc se trouver dans le premier groupe de règles mais nous avons préféré la présenter avec les autres règles relatives aux constructeurs). Elle traduit la construction de l'instance courante dans le cas d'une classe initiale.

La fonction λ_c (cf. définition 1.17) détermine, à partir des types des paramètres de l'appel, le constructeur adéquat et décide donc, notamment, la nouvelle position dans le programme. A partir des informations renvoyées par cette fonction, on construit le nouvel environnement

dont le domaine, comme pour les méthodes, ne reprend que les paramètres formels, les variables locales du constructeur et, bien sûr, **this**.

La différence principale avec les méthodes se situe au niveau de l'image de **this** par cet environnement : on est ici "en train de construire" l'instance courante et, par conséquent, il est normal que celle-ci soit "indéterminée". Plus précisément, on attribue à **this** une nouvelle location dont la valeur est indéterminée.

Au niveau du store, on exprime, comme d'habitude, le passage par valeur des paramètres et la non-initialisation des variables locales.

On stocke sur la pile la variable sur laquelle s'applique l'appel de constructeur afin de savoir, au retour, à qui attribuer la nouvelle instance.

Constr var

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \mathbf{constr}\ x\ nc\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, P + (d_0, lr, (\mathbf{constr}, x)), (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} \lambda_c\ nc\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \bot_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(nc, loc)] \\ l_1, ..., l_n, ll_1, ..., ll_m, loc \notin dom(d_1) \\ d'_1 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}, loc/nc\ \mathbf{ni}] \end{array}\right]$$

La règle *Constr this* n'a de sens que si on vient d'appliquer une autre règle d'appel de constructeur, c'est-à-dire s'il n'existe pas d'instance courante (ce que l'on traduit par une valeur non initialisée pour **this**). A part cela, on retrouve les mécanismes classiques d'un appel (soulignons qu'ici on ne doit pas attribuer de nouvelle location à **this** puisque l'appel dont on vient s'en est déjà chargé).

Constr this

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \mathbf{this}\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, P + (d_0, lr, (\mathbf{constr}, \mathbf{this})), (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(\mathbf{this}))) = (nc, \mathbf{ni}) \\ \lambda_c\ nc\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \bot_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/d_0(\mathbf{this})] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ d'_1 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}] \end{array}\right]$$

Comme pour *Constr this*, la règle *Constr super* n'a de sens que s'il n'existe pas d'instance courante. Cette règle correspond "au début" de la construction de l'instance père de l'instance "en cours de construction". Remarquons, qu'on ne construit jamais une instance avant d'avoir complètement construit ses instances supérieures.

De nouveau, on retrouve les mécanismes classiques d'un appel. On attribue à **this** une nouvelle location qui correspondra, au retour, au pseudo-champ **super** de l'instance "en cours de

construction".

Constr super

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \mathbf{super}\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, P + (d_0, lr, (\mathbf{constr}, \mathbf{super})), (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(\mathbf{this}))) = (nc, \mathbf{ni}) \\ \pi(nc) = np \\ \lambda_c\ np\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(np, loc)] \\ l_1, ..., l_n, ll_1, ..., ll_m, loc \notin dom(d_1) \\ d'_1 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}, loc/np\ \mathbf{ni}] \end{array}\right]$$

Comme on l'a déjà dit, la règle *Constr prem* ne correspond pas du tout à un appel mais se charge directement de la construction de l'instance courante.

Il est donc évident que, comme les deux règles précédentes, cette règle n'est activable que si la valeur de **this** est non initialisée. La fonction Ch nous fournit, à partir du nom de la classe, toutes les informations relatives aux déclarations de champs. A partir de là, on procède "itérativement" :

- on transforme la valeur de **this** en une instance dont le domaine correspond aux champs déclarés et qui associe à chacun de ces champs une nouvelle location de valeur non initialisée,
- on évalue séquentiellement les expressions d'initialisation des champs, en modifiant le store au fur et à mesure.

Constr prem

$$< l, P, (d_0, d_1) > \xrightarrow{l\ \mathbf{prem}\ lr} < lr, P, (d_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(\mathbf{this}))) = (nc, \mathbf{ni}) \\ Ch(nc) = (t_1 ch_1 ex_1 ... t_n ch_n ex_n) \\ s_0 = d_1[l_1/t_1\ \mathbf{ni}, ..., l_n/t_n\ \mathbf{ni}, p_2(d_0(\mathbf{this}))/(nc, v)] \\ l_1, ..., l_n \notin dom(d_1) \\ dom(v) = \{ch_1, ..., ch_n\} \\ \forall i : 1 \leq i \leq n : v(ch_i) = l_i \\ s_1 = s_0[l_1/\mathcal{V}al[\![ex_1]\!]\,(d_0, s_0)] \\ \vdots \\ s_n = s_{n-1}[l_n/\mathcal{V}al[\![ex_n]\!]\,(d_0, d_{n-1})] \\ d'_1 = s_n \end{array}\right]$$

2.2.5 Règles pour les retours

Pour qu'une règle de retour soit activable, il faut que, dans le programme, on atteigne une instruction **return**. Le choix de la règle à activer est alors déterminé par l'information stockée au sommet de la pile.

2.2.5.1 Retour de procédure

Proc retour

$$< l, P + (e, lr, \mathbf{proc}), (d_0, d_1) > \xrightarrow{l\ \mathbf{return}} < lr, P, (e, d_1) >$$

Conditions :

$$\left[\ / \ \right]$$

Lors d'un retour d'un appel de procédure, on reprend juste l'ancien environnement et on "rebranche" au point de programme correspondant à la fin de l'appel.

2.2.5.2 Retour de fonction

Fonc retour

$$< l, P + (e, lr, (\mathbf{fonc}, var)), (d_0, d_1) > \xrightarrow{l\ \mathbf{return}\ ex} < lr, P, (e, d_1[p_2(e(var))/val]) >$$

Conditions :

$$\left[\ \mathcal{V}al[\![ex]\!]\,(d_0, d_1) = val \ \right]$$

Au retour d'un appel de fonction, on reprend l'ancien environnement, on rebranche au point de programme correspondant à la fin de l'appel et on modife le store pour attribuer à la variable stockée la valeur de l'expression mentionnée dans l'instruction **return**.

2.2.5.3 Retour de constructeur

Constr retour

$$< l, P + (e, lr, (\mathbf{constr}, var)), (d_0, d_1) > \xrightarrow{l\ \mathbf{return}} < lr, P, (e, d_1[p_2(e(var))/val]) >$$

Conditions :

$$\left[\ val = d_1(p_2(d_0(\mathbf{this}))) \ \right]$$

Au retour d'un appel de constructeur situé au niveau des instructions, on reprend l'ancien environnement, on rebranche au point de programme correspondant à la fin de l'appel et on modifie le store pour attribuer à la variable stockée l'instance créée.

Constr retour this

$$< l, P + (e, lr, (\mathbf{constr}, \mathbf{this})), (d_0, d_1) > \xrightarrow{l\ \mathbf{return}} < lr, P, (e, d_1) >$$

Conditions :

$$\left[\ / \ \right]$$

Au retour d'un appel à un autre constructeur de la classe, il suffit de reprendre l'ancien environnement (i.e. l'environnement du constructeur en cours) et de reprendre le fil de ce constructeur (i.e. de rebrancher au point de programme correspondant au début du corps du constructeur).

Constr retour super

$$< l, P + (e, lr, (\mathbf{constr}, \mathbf{super})), (d_0, d_1) > \xrightarrow{l\ \mathbf{return}} < lr, P, (d_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(e(\mathbf{this}))) = (nc, \mathbf{ni}) \\ Ch(nc) = (t_1 ch_1 ex_1 ... t_n ch_n ex_n) \\ s_0 = d_1[l_1/t_1\ \mathbf{ni}, ..., l_n/t_n\ \mathbf{ni}, p_2(e(\mathbf{this}))/(nc, v)] \\ l_1, ..., l_n \notin dom(d_1) \\ dom(v) = \{ch_1, ..., ch_n, \mathbf{super}\} \\ \forall i : 1 \leq i \leq n : v(ch_i) = l_i \\ v(\mathbf{super}) = p_2(d_0(\mathbf{this})) \\ s_1 = s_0[l_1/\mathcal{V}al[\![ex_1]\!](d_0, s_0)] \\ \vdots \\ s_n = s_{n-1}[l_n/\mathcal{V}al[\![ex_n]\!](d_0, d_{n-1})] \\ d'_1 = s_n \end{array} \right]$$

Revenir d'un appel de constructeur de l'instance "père" signifie que l'on est "en cours de construction" de l'instance courante. Il faut donc, avant d'entamer le corps du constructeur (en cours), "créer" effectivement l'instance. Pour ce faire, on procède d'une manière tout à fait analogue à celle présentée pour la règle *Constr prem* si ce n'est qu'ici il faut également traiter une instance "père":

- la règle n'est activable que si la valeur de **this** est non initialisée;
- la fonction Ch nous fournit, à partir du nom de la classe, toutes les informations relatives aux déclarations de champs;
- on transforme la valeur de **this** en une instance dont le domaine correspond à l'ensemble des champs déclarés augmenté de **super** et qui associe à chacun de ces champs une nouvelle location de valeur non initialisée et à **super** la location attribuée à **this** dans l'environnement courant,
- on évalue séquentiellement les expressions d'initialisation des champs, en modifiant le store au fur et à mesure.

2.2.6 Etat initial et règle de clôture

On sait qu'il existe une seule classe nc_0 possédant une méthode de signature $main()$.

Si

$$\lambda_m\ nc_0\ main() = (l_0, (), (t_1 w_1 ... t_n w_n)),$$

l'état initial est alors

$$< l_0, (), \perp_0[w_1/t_1 l_1, ..., w_n/t_n l_n, \mathbf{this}/nc_0\ l_{n+1}], \perp_1[l_1/t_1\ \mathbf{ni}, ..., l_m/t_n\ \mathbf{ni}, l_{m+1}/nc_0\ \mathbf{ni})] > .$$

Reste maintenant à définir une dernière règle que nous appelons "règle de clôture". Cette règle est introduite pour traiter l'instruction **return** de l'appel initial à la méthode main (celui-ci n'est en effet pas traité par les autres règles de retour puisqu'aucune d'entre elles n'est activable si la pile est vide).

Clôture

$$< l, (), d > \xrightarrow{l\ \mathbf{return}} d$$

Conditions :

[/]

Les états résultants de l'application de cette règle sont des états "dégénérés", ils n'appartiennent pas à proprement parler à l'ensemble des états tel que nous l'avons défini. Si une exécution aboutit dans un état dégénéré, cela signifie qu'elle se termine sans erreur; inversément, si on arrive dans un état non dégénéré à partir duquel aucune règle ne s'applique, cela signifie qu'on a une erreur d'exécution.

2.2.7 Remarque finale

Pour clore ce chapitre, il faudrait théoriquement vérifier que, pour tous les états manipulés dans ce système de transitions, la troisième composante appartient bien à $\mathbb{D}$, c'est-à-dire qu'il faudrait prouver que chaque règle de transition conserve les quatre propriétés définissantes du domaine d'interprétation.

Cependant, une telle preuve s'avèrerait technique et fastidieuse d'autant que ces propriétés ont en fait été énoncées au vu des différentes transformations possibles du domaine au fil d'une exécution. Nous préférons par conséquent la laisser en suspens pour passer à la suite du travail.

Chapitre 3

Une seconde sémantique

Nous avons défini dans le chapitre 2 une sémantique opérationnelle pour la syntaxe **SAP** introduite au chapitre 1. Cette sémantique se caractérise principalement par deux aspects : sa présentation sous forme d'un système de transitions et son allure naturelle du point de vue de la traduction des mécanismes relatifs aux appels et aux retours de méthodes.

Cette traduction repose sur la présence d'une pile d'environnements dans les états et sur la gestion de celle-ci par les différentes règles de transition. Il nous semble, a priori, qu'au niveau abstrait la gestion de cette pile pourrait s'avérer relativement complexe. C'est pourquoi, nous choisissons, dans ce travail, de définir une autre sémantique qui nous paraît, toujours a priori, plus simple "à abstraire" puisqu'elle ne manipule pas de pile; le prix à payer étant la perte de l'aspect naturel mentionné précédemment.

Cette nouvelle sémantique se présente également sous forme d'un système de transitions : la différence se situe, d'une part, au niveau de la composition des états et, d'autre part, dans l'interprétation à donner à ce système de transitions.

En outre, nous introduisons simultanément une autre modification qui prend place, quant à elle, au niveau du domaine : dans la sémantique précédente, on considérait le store "globalement" et, d'un état à l'autre, on transportait toute l'information de celui-ci; on choisit ici de travailler avec des stores "coupés", c'est-à-dire ne contenant que les valeurs accessibles par l'environnement.

Ce chapitre suit plus ou moins la découpe du chapitre précédent : la première partie, relativement succincte, se consacre à la présentation du domaine tandis que la deuxième commence par expliquer quelle interprétation donner au système de transitions pour ensuite énoncer les règles de celui-ci. On ajoute une troisième partie qui tente de mettre en relation les deux sémantiques présentées.

3.1 Domaine

Cette première partie du chapitre est relativement brève car la plupart des concepts qui s'y rapportent sont repris tels quels du chapitre précédent. On fournit ici deux choses : une nouvelle définition du domaine d'interprétation et deux nouvelles fonctions utiles dans l'expression des règles de transition.

3.1.1 Définition

Les notions de valeurs (cf. définition 2.1), d'environnement et de store (cf. définition 2.2) demeurent naturellement inchangées, de même que les différentes fonctions d'évaluation (cf. définitions 2.7 et 2.8).

La seule différence se situe au niveau de la définition du domaine d'interprétation lui-même.

Le nouveau domaine noté $\mathbb{D}^*$ doit maintenant garantir une propriété supplémentaire pour traduire son aspect "coupé". En incorporant celle-ci aux anciennes propriétés, on obtient les quatre propriétés définissantes suivantes.

Propriété 1 : Forme de l'environnement

$$d_0 = \perp_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \mathbf{this}/t_{n+1}\delta_{n+1}]$$

avec

$$n \in \mathbb{N},\ t_1, ..., t_n \neq \mathbf{bot},\ t_{n+1} \in Nclasse.$$

Propriété 2 : Unicité des locations

$$d_1(l) = (nc, v) \wedge d_1(l') = (nc', v') \Rightarrow (d_1(l) = d_1(l') \vee Im(v) \cap Im(v') = \emptyset).$$

Propriété 3 : Coupure du store

Si on définit la suite de niveaux (D_j) de limite D par

$$D_0 = \{\delta_i \mid 1 \leq i \leq n+1\}$$

$$D_{j+1} = D_j \cup \bigcup_{\substack{l \in D_j \\ d_1(l) \in \mathbb{I}nst}} Im(p_2(d_1(l)))$$

on doit avoir l'égalité

$$D = dom(d_1).$$

Propriété 4 : Respect des types statiques

Si $d_1(l) = (nc, v)$ est un élément de $\mathbb{I}nst$, le store doit vérifier les conditions ci-après.

- **super** $\in dom(v) \Leftrightarrow nc \in dom(\pi)$
- $\forall ch \in Nchamp, ch \in dom(v) \Leftrightarrow ch \in dom(\epsilon^0_{ch} nc)$
- **super** $\in dom(v) \Rightarrow d_1(v(\textbf{super}))) = (\pi(nc), p) \in \mathbb{I}nst$
- $ch \in dom(v) \Rightarrow p_1(d_1(v(ch))) \preceq_\pi (\epsilon^0_{ch}\, nc\, ch)$

On a également

$$\forall i : 1 \leq i \leq n : p_1(d_1(\delta_i)) \preceq_\pi t_i,$$

$$p_1(d_1(\delta_{n+1})) = t_{n+1}.$$

Définition 3.1 (Domaine $\mathbb{D}^*$)

$\mathbb{D}^*$ désigne le domaine d'interprétation et est défini par la relation d'appartenance suivante. Le couple $d = (d_0, d_1)$ appartient à $\mathbb{D}^*$ si et seulement si d appartient au produit cartésien $\mathbb{E}nv \times \mathbb{S}tore$ et vérifie les quatre propriétés 1, 2, 3 et 4.

Même si on ne le mentionne plus, ce domaine est, tout comme son prédécesseur, dépendant du programme considéré.

3.1.2 Deux nouvelles fonctions

Les deux définitions introduites ici décrivent en fait le même genre de concept : elles fournissent le même type de résultat mais à partir d'informations de types différents. Elles interviennent dans l'expression des règles de transition et sont reliées, notamment, à l'aspect "coupé" des domaines.

La première, nommée *Attloc*, détecte les locations accessibles à partir d'une location donnée dans un store donné. La seconde, nommée *Attval*, identifie les locations attachées à une valeur donnée (i.e. intervenant dans la structure correspondant à cette valeur) dans un store donné.

Définition 3.2 (Fonction *Attloc*)

La fonction Attloc renvoie l'ensemble des locations attachées à une location donnée dans un store donné.

$$\mathcal{A}ttloc : \$tore \longrightarrow \mathbb{L}oc \longrightarrow \wp(\mathbb{L}oc)$$

$$\mathcal{A}ttloc\ s\ l = \emptyset \quad \textbf{si}\ l \notin dom(s)$$
$$\mathcal{A}ttloc\ s\ l = \emptyset \quad \textbf{si}\ s(l) \in \mathbb{B}ase$$
$$\mathcal{A}ttloc\ s\ l = \bigcup\{\mathcal{A}ttloc\ s\ l' \mid l' \in Im(p_2(s(l)))\} \cup Im(p_2(s(l))) \quad \textbf{si}\ s(l) \in \mathbb{I}nst$$

Remarquons que l'ensemble renvoyé par cette fonction ne contient pas la location de départ.

On peut reformuler la propriété de coupure du store en terme de cette fonction :

$$dom(d_1) = \{\delta_1, ..., \delta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ d_1\ \delta_i.$$

Définition 3.3 (Fonction *Attval*)

La fonction Attval renvoie l'ensemble des locations attachées à une valeur donnée dans un store donné.

$$\mathcal{A}ttval : \$tore \longrightarrow \mathbb{V}al \longrightarrow \wp(\mathbb{L}oc)$$

$$\mathcal{A}ttval\ s\ v = \emptyset \quad \textbf{si}\ v \in \mathbb{B}ase$$
$$\mathcal{A}ttval\ s\ v = \bigcup\{\mathcal{A}ttval\ s\ s(l') \mid l' \in Im(p_2(v))\} \cup Im(p_2(v)) \quad \textbf{si}\ s(l) \in \mathbb{I}nst$$

3.2 Système de transitions

Cette section présente le système de transitions correspondant à la seconde sémantique. On commence par définir quels sont les états de celui-ci. On précise ensuite quelle interprétation lui donner pour enfin exposer les règles de transition elles-mêmes. Tout au long de l'énoncé de ces règles, on tentera d'effectuer des comparaisons avec les règles du système précédent.

3.2.1 Etats

Le but avoué de cette sémantique est d'éliminer la pile des environnements présente dans les états. La forme des états se simplifie donc, ceux-ci ne reprenant plus qu'un point de programme et un élément du domaine d'interprétation.

Définition 3.4 (Ensemble des états)

Ɛtat dénote l'ensemble des états du système de transitions.

$$Ɛtat = Lab \times \mathbb{D}^*$$

Il est évident que cette simplification ne peut qu'entraîner des complications au niveau des règles de transition. En anticipant sur la suite, on peut en effet s'interroger sur l'expression des règles de retour puisque celles-ci se basaient, dans la première sémantique, sur l'information du sommet de la pile.

3.2.2 Interprétation

Dans le système précédent, toutes les règles de transition respectaient le format

Nom de règle

$$e \xrightarrow{Morceau\ de\ programme} e'$$

Conditions :

[*Liste de conditions*]

Il se trouve que, vu notre définition des états, on ne peut plus se contenter de règles de cette forme. En effet, un seul état ne contient plus toute l'information nécessaire pour décrire le mécanisme lié au retour d'un appel de méthode. Pour pallier à cette carence, on introduit, pour traduire les règles de retour, le nouveau format de règle décrit ci-après.

Nom de règle

$$< e, e' > \xrightarrow[instruction\ d'appel]{instruction\ de\ retour} e''$$

Conditions :

[*Liste de conditions*]

Puisque ces règles partent de deux états au lieu d'un seul, on parlera de *règles conjointes.*

Comment interpréter ce type de règle ? Intuitivement, on désire qu'une telle règle exprime le passage de l'état e avant l'appel *instruction d'appel* à l'état e'' après le même appel, sachant que l'état e' correspond à l'état obtenu à la fin du corps de la méthode (en partant de e).

Pour pouvoir interpréter ces règles, il nous faut revoir le sens attribué à un système de transitions $< E, e_0, (\mathcal{R}_i)_{1 \leq i \leq k} >$: il ne s'agit plus de décrire pas à pas des exécutions mais plutôt de générer un ensemble d'états "accessibles".

Ainsi les règles ne s'interprètent plus comme des fonctions de l'ensemble des états dans lui-même mais comme des transformations de l'ensemble des parties de celui-ci (remarquons qu'on aurait tout à fait pu interpréter l'ancien système de transitions selon la même optique).

Une règle simple, i.e. suivant le premier format, $\mathcal{R}_i$ s'interprète comme une fonction de la forme

$$\begin{array}{rccl} \mathcal{R}_i : & \wp(E) & \longrightarrow & \wp(E) \\ & W & \rightsquigarrow & W \cup \{e' \mid e \in W \wedge e \text{ et } e' \text{ vérifient les conditions d'activation de } \mathcal{R}_i\}. \end{array}$$

Une règle conjointe, i.e. suivant le second format, $\mathcal{R}_j$ s'interprète comme une fonction de la forme

$$\begin{array}{rccl} \mathcal{R}_j : & \wp(E) & \longrightarrow & \wp(E) \\ & W & \rightsquigarrow & W \cup \{e'' \mid e, e' \in W \wedge (e, e', e'') \text{ vérifie les conditions d'activation de } \mathcal{R}_j\}. \end{array}$$

On peut maintenant passer à la description des différentes règles de transition. On conserve la répartition en *Règles simples*, *Règles relatives aux appels* et *Règles relatives aux retours*. Les règles conjointes interviennent naturellement uniquement dans le troisième groupe.

3.2.3 Règles simples

Puique la pile n'intervenait pas dans ces règles, il paraît logique que celles-ci ne changent pas si ce n'est au niveau de la forme des états. On doit quand même ajouter des conditions à la règle concernant l'affectation pour traduire la conservation de l'aspect "coupé" du store.

On fournit donc seulement cette dernière règle.

Affect

$$< l, (d_0, d_1) > \xrightarrow{l \textbf{ affect } des\ expr\ l'} < l', (d_0, d_1') >$$

Conditions :

$$\left[\begin{array}{l} \mathcal{A}dr[\![des]\!]\,(d_0, d_1) = loc \\ \mathcal{V}al[\![expr]\!]\,(d_0, d_1) = val \\ s = d_1[loc/val] \\ Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d_0)\} \\ d_1' = s_{|_{Cut}} \end{array}\right]$$

Si f est une fonction, $f_{|_E}$ désigne la fonction obtenue en restreignant le domaine de f à l'ensemble E.

3.2.4 Règles pour les appels

3.2.4.1 Appels de procédure

Les règles concernant les appels de procédure diffèrent des règles de la première sémantique en deux points :

- puisqu'il n'existe plus de pile dans les états, il n'y a plus moyen de stocker d'information

relative au contexte de l'appel dans l'état résultant;

- on doit garantir l'aspect "coupé" du store également pour l'état résultant (i.e. l'état correspondant au début du corps de la procédure).

Les transformations étant analogues pour les trois règles, on ne développe ici que l'une d'entre-elles.

Proc var

$$< l, (d_0, d_1) > \xrightarrow{l\ \mathbf{proc}\ (tr\ var\ meth\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(var))) = (nc, v) \in \mathit{Inst} \\ \lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \bot_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(nc, p_2(d_0(var)))] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ s = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}] \\ Cut = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d'_0)\} \\ d'_1 = s_{|Cut} \end{array}\right]$$

3.2.4.2 Appels de fonctions

Les règles traitant les appels de fonction subissent des transformations tout à fait similaires aux transformations évoquées pour les appels de procédure.

Remarquons que comme il n'y a plus d'information particulière à sauvegarder, ces règles deviennent, au morceau de programme traité près, identiques aux règles pour les procédures.

Ces règles sont au nombre de trois : *Fonc var*, *Fonc this*, *Fonc super*.

On donne, à titre d'illustration, la règle *Fonc this*.

Fonc this

$$< l, (d_0, d_1) > \xrightarrow{l\ \mathbf{fonc}\ x\ (tr\ \mathbf{this}\ meth\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(\mathbf{this}))) = (nc, v) \in \mathit{Inst} \\ \lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \bot_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/d_0(\mathbf{this})] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ s = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}] \\ Cut = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d'_0)\} \\ d'_1 = s_{|Cut} \end{array}\right]$$

3.2.4.3 Appels de constructeurs

De nouveau, on applique les mêmes transformations sur les règles *Constr var*, *Constr this* et *Constr super*. La règle *Constr prem* est, quant à elle, complètement inchangée.

3.2.5 Règles pour les retours

C'est évidemment à ce niveau qu'interviennent les principales modifications par l'introduction de règles conjointes.

L'expression de ces règles conjointes nécessite d'exprimer des conditions d'activation reliant non plus deux états mais bien trois : si $< p, d >$ désigne l'état avant l'appel, on doit, dans un premier temps, exprimer des conditions de "compatibilité" entre celui-ci et un autre état $< l, d' >$ censé décrire un état obtenu à partir de $< p, d >$ à la fin de l'exécution du corps de la méthode/du constructeur appelé. On peut alors seulement rédiger des conditions définissant l'état résultat.

Intuitivement, on sent que les conditions de "compatibilité" ne peuvent qu'éliminer du couple initial de la règle des associations que nous estimons inacceptables, mais ne peuvent pas garantir que certains couples activant la règle ne l'active pas "par hasard", c'est-à-dire qu'il se peut que l'état $< l, d' >$ soit compatible avec l'état $< p, d >$ mais qu'effectivement il n'ait pas été engendré par "l'exécution" de l'appel dérivant de $< p, d >$.

Dans ce sens, on dira que cette seconde sémantique est "une approximation" de la première.

3.2.5.1 Retour de procédure

On avait, pour la première sémantique, une seule règle pour les retours de procédure. On en obtient, dans ce cas, trois, c'est-à-dire autant qu'il y a de formes d'appel de procédure. Ces règles, nommées *Proc var retour*, *Proc this retour* et *Proc super retour* étant quasi identiques, on choisit de n'en développer qu'une seule à savoir *Proc var retour*.

Avant d'énoncer cette règle, précisons une nouvelle notation employée dans celle-ci.

Si f et g dénotent des fonctions de signature $A \longmapsto B$ et E un sous-ensemble de A, $f[E/g]$ dénote la fonction suivante.

$$
\begin{array}{lllll}
f[E/g]: & A & \longmapsto & B & \\
 & a & \rightsquigarrow & g(a) & \textbf{si } a \in E \\
 & a & \rightsquigarrow & f(a) & \textbf{si } a \notin E
\end{array}
$$

Proc var retour

$$<< p, (d_0, d_1) >, < l, (d'_0, d'_1) >> \xrightarrow[p \textbf{ proc } (tr\ var\ meth\ (t_1 e_1 ... t_n e_n))\ lr]{l \textbf{ return}} < lr, (d_0, d^r_1) >$$

Conditions :

Conditions de compatibilité
$d_1(p_2(d_0(var))) = (nc, v) \in \mathit{Inst}$
$\lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m))$
$d'_0 = \bot_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \textbf{this}/(nc, p_2(d_0(var)))]$
$l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1)$
$s_0 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \textbf{ni}, ..., ll_m/tt_m\ \textbf{ni}]$
$Cut_0 = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s_0\ l \mid l \in Im(d'_0)\}$
$l \in dom(d'_1) \setminus Cut_0 \Rightarrow l \notin dom(d_1)$
$d'_1(p_2(d'_0(\textbf{this}))) = s_0(p_2(d'_0(\textbf{this})) \wedge l <_\pi p_2(d'_0(\textbf{this})) \Rightarrow d'_1(l) = s_0(l)$

Conditions sur le résultat
$Int = \{p_2(d'_0(\textbf{this}))\} \cup \mathcal{A}ttloc\ d'_1\ p_2(d'_0(\textbf{this})) \cup \bigcup_{i=1}^{n} \mathcal{A}ttloc\ d'_1\ l_i$
$s = d_1[Int/d'_1]$
$Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d_0)\}$
$d^r_1 = s_{|_{Cut}}$

Dans la règle ci-avant, $< p, (d_0, d_1) >$ désigne l'état avant l'appel et $< l, (d'_0, d'_1) >$ l'état supposé correspondre à la fin du corps de la méthode appelée.

Exprimer les conditions de compatibilité entre ces deux états nécessite de refaire, au moins partiellement, le travail effectué par la règle *Proc var* : en effet, dans un premier temps, on reconstruit l'état correspondant au début du corps de la méthode, soit $< ldeb, (d'_0, s_{0|_{Cut_0}}) >$.

Pour qu'on puisse, à la fin de l'appel, "recoller" les informations du store de la procédure dans le store de l'appel, il faut que "ce qu'on recolle" n'interfère pas avec ce qui, dans le store de l'appel, n'est pas concerné par la méthode. On exprime cette idée en exigeant que toute location de d'_1 non nécessaire à l'expression des différents paramètres au moment de l'appel soit libre dans d_1.

On complète les conditions de compatibilité par une propriété exprimant la conservation de la structure de l'instance courante tout au long de l'exécution du corps de la méthode.

Les conditions sur le résultat précisent d'abord quels sont les "morceaux" de d'_1 qui doivent être "insérés" dans d_1 (il s'agit naturellement des valeurs "complètes" des paramètres, de la location associée à la variable sur laquelle se réalise l'appel et de la valeur complète de celle-ci) et, comme d'habitude, restreignent le store résultat aux locations accessibles via l'environnement.

3.2.5.2 Retour de fonction

Comme pour les retours de procédure, on a maintenant trois règles au lieu d'une : *Fonc this retour*, *Fonc super retour* et *Fonc var retour*. On ne développe ici que *Fonc this retour*.

Les conditions de compatibilité sont identiques à celles présentées pour les procédures.

La seule différence de traitement émerge au moment du "recollage" de d'_1 dans d_1. En effet, il faut ici insérer une partie plus étendue de d'_1 puisqu'il faut tenir compte des locations intervenant dans la valeur renvoyée par la fonction.

Fonc this retour

$$<< p, (d_0, d_1) >, < l, (d'_0, d'_1) >> \xrightarrow[p\ \mathbf{fonc}\ x\ (tr\ var\ meth\ (t_1 e_1 ... t_n e_n))\ lr]{l\ \mathbf{return}\ ex} < lr, (d_0, d^r_1) >$$

Conditions :

Conditions de compatibilité

$d_1(p_2(d_0(\mathbf{this}))) = (nc, v) \in \mathbb{I}nst$

$\lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m))$

$d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/d_0(\mathbf{this})]$

$l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1)$

$s_0 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}]$

$Cut_0 = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s_0\ l \mid l \in Im(d'_0)\}$

$l \in dom(d'_1) \setminus Cut_0 \Rightarrow l \notin dom(d_1)$

$d'_1(p_2(d'_0(\mathbf{this}))) = s_0(p_2(d'_0(\mathbf{this})) \wedge l <_\pi p_2(d'_0(\mathbf{this})) \Rightarrow d'_1(l) = s_0(l)$

Conditions sur le résultat

$Int = \{p_2(d'_0(\mathbf{this}))\} \cup \mathcal{A}ttloc\ d'_1\ p_2(d'_0(\mathbf{this}))$
$\cup \mathcal{A}ttval\ d'_1\ (\mathcal{V}al[\![ex]\!](d'_0, d'_1)) \cup \bigcup_{i=1}^{n} \mathcal{A}ttloc\ d'_1\ l_i$

$s = (d_1[Int/d'_1])[p_2(d_0(x))/\mathcal{V}al[\![ex]\!](d'_0, d'_1)]$

$Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d_0)\}$

$d^r_1 = s_{|Cut}$

3.2.5.3 Retour de constructeur

On conserve le même nombre de règles d'une sémantique à l'autre, c'est-à-dire trois. La règle *Constr retrour* correspond toujours au retour d'un constructeur situé au niveau des instructions; la règle *Constr retour this* correspond au retour d'un autre constructeur de la classe et la règle *Constr retour super* correspond au retour d'un constructeur de la classe "père".

De nouveau, pour ces trois règles, les conditions de compatibilité répètent le travail de leur homologue au niveau des appels.

Commençons par la règle *Constr retour.*

Constr retour

$$<< p, (d_0, d_1) >, < l, (d'_0, d'_1) >> \xrightarrow[p\ \mathbf{constr}\ x\ nc\ (t_1 e_1 ... t_n e_n))\ lr]{l\ \mathbf{return}} < lr, (d_0, d^r_1) >$$

Conditions :

Conditions de compatibilité

$\lambda_c\ nc\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m))$
$d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(nc, loc)]$
$l_1, ..., l_n, ll_1, ..., ll_m, loc \notin dom(d_1)$
$s_0 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}, loc/nc\ \mathbf{ni}]$
$Cut_0 = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s_0\ l \mid l \in Im(d'_0)\}$
$l \in dom(d'_1) \setminus Cut_0 \Rightarrow l \notin dom(d_1)$
$d'_1(loc) = (nc, v) \in \mathbb{I}nst$

Conditions sur le résultat

$Int = \mathcal{A}ttloc\ d'_1\ loc \cup \bigcup_{i=1}^{n} \mathcal{A}ttloc\ d'_1\ l_i$
$s = (d_1[Int/d'_1])[d_0(x)/d'_1(loc)]$
$Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d_0)\}$
$d^r_1 = s_{|_{Cut}}$

Dans cette règle, $< p, (d_0, d_1) >$ désigne, comme toujours, l'état avant l'appel et $< l, (d'_0, d'_1) >$ l'état supposé correspondre à la fin du corps du constructeur appelé. L'état correspondant au début du corps de la méthode est alors désigné par $< ldeb, (d'_0, s_{0_{|_{Cut_0}}}) >$.

Il faut de nouveau, qu'à la fin de l'appel, les parties de d'_1 "recollées" dans d_1 n'interfèrent pas avec ce qui n'est pas modifiable par la méthode.

La propriété de conservation de la structure d'extension de l'instance courante n'a bien sûr plus aucun sens ici, puisque justement l'appel est supposé avoir construit une nouvelle instance. On exige simplement ici que le store fasse effectivement correspondre une instance à la nouvelle location *loc* attribuée à **this** dans l'environnement du constructeur.

Comme pour les méthodes, les conditions sur le résultat précisent d'abord les "morceaux" de d'_1 qui doivent être "insérés" dans d_1 et restreignent ensuite le store résultat aux locations accessibles via l'environnement. On doit évidemment ici attribuer à la variable sur laquelle se réalise l'appel l'instance construite.

Enonçons maintenant la règle *Constr retour this.*

Constr retour this

$$<< p, (d_0, d_1) >, < l, (d'_0, d'_1) >> \xrightarrow[p\ \mathbf{this}\ (t_1 e_1 ... t_n e_n)\ lr]{l\ \mathbf{return}} < lr, (d_0, d^r_1) >$$

Conditions :

Conditions de compatibilité

$d_1(p_2(d_0(\mathbf{this}))) = (nc, \mathbf{ni})$
$\lambda_c\ nc\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m))$
$d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/d_0(\mathbf{this})]$
$l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1)$
$s_0 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}]$
$Cut_0 = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s_0\ l \mid l \in Im(d'_0)\}$
$l \in dom(d'_1) \setminus Cut_0 \Rightarrow l \notin dom(d_1)$
$d'_1(p_2(d'_0(\mathbf{this}))) = (nc, v) \in \mathbb{I}nst$

Conditions sur le résultat

$Int = \{p_2(d'_0(\mathbf{this}))\} \cup \mathcal{A}ttloc\ d'_1\ p_2(d'_0(\mathbf{this})) \cup \bigcup_{i=1}^{n} \mathcal{A}ttloc\ d'_1\ l_i$
$s = d_1[Int/d'_1]$
$Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d_0)\}$
$d^r_1 = s_{|Cut}$

Cette règle est presque identique à la précédente : la seule différence se situe au niveau de l'affectation finale de l'instance construite.

On peut maintenant aborder notre dernière règle : *Constr retour super.*

Constr retour super

$$<< p, (d_0, d_1) >, < l, (d'_0, d'_1) >> \xrightarrow[p\ \mathbf{super}\ (t_1 e_1 ... t_n e_n)\ lr]{l\ \mathbf{return}} < lr, (d_0, d^r_1) >$$

Conditions :

Conditions de compatibilité

$d_1(p_2(d_0(\mathbf{this}))) = (nc, \mathbf{ni})$
$\pi(nc) = np$
$\lambda_c\ np\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m))$
$d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(np, loc)]$
$l_1, ..., l_n, ll_1, ..., ll_m, loc \notin dom(d_1)$
$s_0 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\ (d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\ (d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}, loc/np\ \mathbf{ni}]$
$Cut_0 = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s_0\ l \mid l \in Im(d'_0)\}$
$l \in dom(d'_1) \setminus Cut_0 \Rightarrow l \notin dom(d_1)$
$d'_1(loc) = (np, f) \in \mathbb{I}nst$

Conditions sur le résultat

$Int = \mathcal{A}ttloc\ d'_1\ loc \cup \bigcup_{i=1}^{n} \mathcal{A}ttloc\ d'_1\ l_i$
$Ch(nc) = (\tau_1\ ch_1\ ex_1 ... \tau_k\ ch_k\ ex_k)$
$s = d_1[Int/d'_1]$
$\sigma_0 = s[loc_1/\tau_1\ \mathbf{ni}, ..., loc_k/\tau_k\ \mathbf{ni}, p_2(d_0(\mathbf{this}))/(nc, v)]$
$dom(v) = \{ch_1, .., ch_k, \mathbf{super}\}$
$\forall i : 1 \leq i \leq k : v(ch_i) = loc_i$
$v(\mathbf{super}) = loc$
$\sigma_1 = \sigma_0[loc_1/\mathcal{V}al[\![ex_1]\!]\ d_0\ \sigma_0]$
$\vdots$
$\sigma_k = \sigma_{k-1}[loc_k/\mathcal{V}al[\![ex_k]\!]\ d_0\ \sigma_{k-1}]$
$Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ \sigma_k\ l \mid l \in Im(d_0)\}$
$d^r_1 = (\sigma_k)_{|Cut}$

Les conditions de compatibilité sont très similaires à celles présentées pour les autres règles. Les conditions sur le résultat doivent, comme pour la règle de la première sémantique, exprimer la construction effective de l'instance courante à partir, notamment de l'instance "père" (np, f) supposée fournie par le store correspondant à la fin du corps du constructeur d'_1.

3.2.6 Etat initial et règle de clôture

L'état initial est quasi identique à celui de la première sémantique (évidemment, il ne reprend pas de pile), c'est à dire que si

$$\lambda_m\ nc_0\ main() = (l_0, (), (t_1 w_1 ... t_n w_n)),$$

l'état initial est le couple $< l_0, (e, s) >$ où

$$e = \perp_0[w_1/t_1 l_1, ..., w_n/t_n l_n, \mathbf{this}/nc_0\, l_{n+1}],$$
$$s = \perp_1[l_1/t_1\ \mathbf{ni}, ..., l_m/t_n\ \mathbf{ni}, l_{m+1}/nc_0\ \mathbf{ni})] > .$$

Pour générer, l'ensemble des états accessibles on commencera donc par appliquer l'ensemble des règles au singleton $\{< l_0, (e, s) >\}$.

Pour traiter le "retour" de la méthode *main*, on introduit finalement la règle de clôture suivante.

Clôture

$$<< l_0, (e, s) >, < l, (e, s') >> \xrightarrow{l\ \mathbf{return}} (e, s')$$

Conditions :

[/]

Cette règle permet en outre, dans les cas où le programme se termine, de déterminer si le programme s'achève sans erreur.

3.3 Lien entre les deux sémantiques

3.3.1 Approximation

Nous avons signalé, lors de l'explication des règles de transition, que la sémantique proposée dans ce chapitre n'était pas équivalente à la sémantique fournie dans le chapitre précédent mais qu'elle "approximait" cette dernière.

Il nous semble souhaitable de formuler cette affirmation un peu plus précisément. C'est ce à quoi nous nous attelons dans cette section. Il est évident qu'il serait tout aussi souhaitable de prouver cette affirmation mais nous n'avons malheureusement pas le temps de nous attaquer à cette tâche.

Notons $T =< E, e_0, (\mathcal{R}_i)_{1 \leq i \leq k} >$ le système de transitions décrivant la première sémantique et $T' =< E', e'_0, (\mathcal{R}'_i)_{1 \leq i \leq k'} >$ le système de transitions correspondant à la seconde.

Une des caractéristiques fondamentales de T est son déterminisme. Il est clair que cette caractéristique n'est pas partagée par T'. Rappelons ce que nous entendons par exécution de T et définissons ce qu'est l'exécution de T'.

Dans T, les règles de transition sont interprétées comme des fonctions partielles de la forme

$$\mathcal{R}_i : E \nrightarrow E.$$

L'éxécution de T est alors la suite d'états

$$e_0 \rightarrow e_1 \rightarrow ... \rightarrow e_n \rightarrow ...$$

telle que

$$\forall i \geq 0, \exists k \text{ tel que } \mathcal{R}_k(e_i) = e_{i+1}.$$

Cette suite d'états peut être finie ou infinie.

Dans T', les règles de transition sont interprétées comme des fonctions de la forme

$$\mathcal{R}_i : \wp(E) \longrightarrow \wp(E).$$

L'exécution de T' est alors la suite d'ensembles d'états

$$W_0 = \{e_0\} \subseteq W_1 \subseteq ... \subseteq W_n \subseteq ...$$

telle que

$$\forall i \geq 0, W_{i+1} = \bigcup_{j=1}^{k'} \mathcal{R}_j(W_i).$$

Définissons maintenant une fonction permettant de transformer un état de E en un état de E'. Cette fonction doit tout simplement enlever la pile et "couper" le store :

$$\begin{array}{cccc} \mu : & E & \longrightarrow & E \\ & < l, P, (d_0, d_1) > & \rightsquigarrow & < l, (d_0, d'_1) > \end{array}$$

où $d'_1 = d_{1|_{Cut}}$ et $Cut = Im(d_0) \cup \bigcup\{Attloc\ d_1\ l \mid l \in Im(d_0)\}$.

A partir de ces différentes définitions, l'approximation de la première sémantique par la seconde s'exprime succinctement sous la forme :

$$\forall n,\ \mu(e_n) \in W_n$$

3.3.2 Elimination du "bruit"

Dans cette dernière section, nous proposons une "amélioration" de la seconde sémantique permettant d'éliminer le "bruit", i.e. l'imprécision, introduit par celle-ci. Nous donnons juste le principe de cette "amélioration".

L'introduction du bruit se situe au niveau des règles conjointes, i.e. des règles de retour. Le problème rencontré lors de la rédaction de ces règles était de définir précisément une notion "d'états compatibles". L'idéal serait de pouvoir identifier un seul état compatible qui correspondrait naturellement à l'appel "dont on revient". On rendrait alors le système de transitions déterministe.

Il est évident que l'information conservée dans les états tels qu'il sont définis jusqu'à présent n'est pas suffisante pour réaliser cela. L'idée est alors d'ajouter une "horloge" au système de transitions. Cette horloge compterait le nombre "actions élémentaires" réalisées au cours d'une

exécution et permettrait donc d'identifier de manière univoque chaque état au cours de celle-ci, en particulier les états correspondant aux appels. Pour déterminer lors des retours l'appel correspondant, on stockerait également dans chaque état la valeur de l'horloge identifiant l'appel en cours.

Illustrons le principe donné ci-dessus en fournissant, d'une part, une nouvelle définition pour les états et, d'autre part, en énonçant quelques règles de transition (on donne une règle simple, une règle d'appel et une règle de retour).

L'ensemble des états est égal au produit cartésien $Lab \times I\!N \times I\!N \times I\!D^*$. Si un état est de la forme $< l, h, a, d >$, le composant h fournit la valeur de l'horloge correspondant au temps où cet état a été généré tandis que a identifie l'appel en cours.

Une règle simple incrémente juste l'horloge et ne modifie naturellement pas l'appel en cours.

Skip

$$< l, h, a, (d_0, d_1) > \xrightarrow{l\ \mathbf{skip}\ l'} < l', h+1, a, (d_0, d'_1) >$$

Conditions :

[/]

Une règle d'appel incrémente l'horloge et mémorise le temps correspondant à l'appel.

Proc var

$$< l, h, a, (d_0, d_1) > \xrightarrow{l\ \mathbf{proc}\ (tr\ var\ meth\ (t_1 e_1 ... t_n e_n))\ lr} < ldeb, h+1, h, (d'_0, d'_1) >$$

Conditions :

$$\left[\begin{array}{l} d_1(p_2(d_0(var))) = (nc, v) \in I\!Inst \\ \lambda_m\ nc\ meth\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m)) \\ d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(nc, p_2(d_0(var)))] \\ l_1, ..., l_n, ll_1, ..., ll_m \notin dom(d_1) \\ s = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}] \\ Cut = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d'_0)\} \\ d'_1 = s_{|Cut} \end{array}\right]$$

Une règle conjointe ne peut s'appliquer à un état correspondant à un appel et à un état correspondant à la fin d'un corps de méthode ou de constructeur que si l'appel en cours dans le second correspond à la valeur de l'horloge pour le premier. En ce qui concerne l'état résultant (i.e. l'état correspondant au retour de l'appel), on incrémente l'horloge du second état et on reprend l'appel en cours dans le premier état.

Constr retour

$$<< p, h, a, (d_0, d_1) >, < l, h', h, (d'_0, d'_1) >> \xrightarrow[p\ \mathbf{constr}\ x\ nc\ (t_1 e_1 ... t_n e_n))\ lr]{l\ \mathbf{return}} < lr, h' + 1, a, (d_0, d^r_1) >$$

Conditions :

Conditions de compatibilité

$\lambda_c\ nc\ (t_1 ... t_n) = (ldeb, (t'_1 v_1 ... t'_n v_n), (tt_1 w_1 ... tt_m w_m))$
$d'_0 = \perp_0[v_1/t'_1 l_1, ..., v_n/t'_n l_n, w_1/tt_1 ll_1, w_m/tt_m ll_m, \mathbf{this}/(nc, loc)]$
$l_1, ..., l_n, ll_1, ..., ll_m, loc \notin dom(d_1)$
$s_0 = d_1[l_1/\mathcal{V}al[\![t_1 e_1]\!]\,(d_0, d_1), ..., l_n/\mathcal{V}al[\![t_n e_n]\!]\,(d_0, d_1), ll_1/tt_1\ \mathbf{ni}, ..., ll_m/tt_m\ \mathbf{ni}, loc/nc\ \mathbf{ni}]$
$Cut_0 = Im(d'_0) \cup \bigcup\{\mathcal{A}ttloc\ s_0\ l \mid l \in Im(d'_0)\}$
$l \in dom(d'_1) \setminus Cut_0 \Rightarrow l \notin dom(d_1)$
$d'_1(loc) = (nc, v) \in \mathbb{I}nst$

Conditions sur le résultat

$Int = \mathcal{A}ttloc\ d'_1\ loc \cup \bigcup_{i=1}^{n} \mathcal{A}ttloc\ d'_1\ l_i$
$s = (d_1[Int/d'_1])[d_0(x)/d'_1(loc)]$
$Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ s\ l \mid l \in Im(d_0)\}$
$d^r_1 = s_{|Cut}$

Cette nouvelle sémantique devrait quant à elle permettre d'aboutir à un résultat d'équivalence entre celle-ci et la "sémantique naturelle" du chapitre précédent.

En introduisant les fonctions de transformation

$$\begin{array}{rccc} \mu_2 : & E' & \longrightarrow & Lab \times \mathbb{D}^* \\ & < l, h, a, (d_0, d_1) > & \rightsquigarrow & < l, (d_0, d_1) > \end{array}$$

et

$$\begin{array}{rccc} \mu_1 : & E & \longrightarrow & Lab \times \mathbb{D}^* \\ & < l, P, (d_0, d_1) > & \rightsquigarrow & < l, (d_0, d'_1) >, \end{array}$$

où $d'_1 = d_{1|Cut}$ et $Cut = Im(d_0) \cup \bigcup\{\mathcal{A}ttloc\ d_1\ l \mid l \in Im(d_0)\}$,

on peut formuler ce résultat d'équivalence par exemple sous la forme

$$\forall n,\ \mu_2(W_{n+1}) = \mu_2(W_n) \cup \{\mu_1(e_{n+1})\}.$$

Chapitre 4

Domaine abstrait

Dans ce chapitre, nous abordons la partie "analyse" à proprement parler. Notre objectif est à présent de rédiger une sémantique abstraite approximant la sémantique présentée au chapitre précédent, tout en gardant à l'esprit que l'information principale qui nous intéresse porte sur les types des différents éléments du programme. Plus précisément, on aimerait, en chaque point de programme et pour chaque variable (au sens large), pouvoir fournir avec certitude "l'ensemble" des types dynamiques possibles pour cette variable.

Nous choisissons ici d'opter pour une sémantique abstraite "homomorphique" à la sémantique concrète. Par conséquent, cette sémantique abstraite revêtra, tout comme la sémantique concrète, la forme d'un système de transitions. Il faudra abstraire, d'une part, les états du système et, d'autre part, les différentes règles de transition de celui-ci.

Ce chapitre se situe au niveau de l'abstraction des états et propose une première définition du domaine abstrait. Pour pouvoir par la suite construire l'équivalent abstrait des règles de transition, il faut que l'on puisse réaliser quelques opérations classiques sur ce domaine. Nous nous contentons ici d'aborder le problème de la borne supérieure.

Nous suivons plus ou moins, durant ce chapitre, le cheminement suivant : pour commencer, nous précisons les propriétés souhaitables du domaine abstrait primitif (i.e. du domaine rassemblant les informations abstraites "de base"); nous définissons ensuite, à partir de celui-ci, notre domaine abstrait; nous fournissons les fonctions de concrétisation et d'abstraction relatives à ce domaine et, après l'introduction d'une structure de préordre sur le domaine, nous discutons de la possibilité du calcul d'un opérateur de borne supérieure sur celui-ci. Cette discussion nous amène finalement à proposer une version quelque peu modifiée du domaine.

Remarquons que cette présentation se rapproche du cadre théorique habituel pour l'interprétation abstraite, à savoir les "insertions de Galois", sans "adhérer" complètement à celui-ci. En effet, une "insertion de Galois" manipule d'une part une fonction de concrétisation et d'autre part une fonction d'abstraction. Une telle structure nécessite en outre la présence d'ordres sur les domaines abstraits et concrets et requiert la monotonie de la concrétisation relativement à l'ordre abstrait. Mais la caractéristique principale de "l'insertion de Galois" est sans doute la détermi-

nation réciproque d'une des fonctions par l'autre.

Une première différence réside ici dans la manipulation d'un préodre. Au niveau théorique, cette différence n'est cependant pas cruciale, le passage "au quotient" permettant d'obtenir à partir de celui-ci un ordre. La différence essentielle se situe au niveau de la détermination réciproque des fonctions : la fonction d'abstraction intervenant dans une "insertion de Galois" est en fait une fonction d'abstraction pour un ensemble d'éléments concrets, or nous ne donnons ici que la fonction d'abstraction d'un seul élément concret. L'extension d'une telle fonction à l'ensemble des parties passe usuellement par la définition d'un opérateur de borne supérieure, or, a priori, nous ne pouvons garantir l'existence d'un tel opérateur.

4.1 Domaine abstrait primitif

Le domaine abstrait primitif doit contenir les informations abstraites élémentaires. Il est évident que, dans le contexte de l'analyse choisie, celles-ci sont intimement liées à des notions de types.

Nous désignons ce domaine primitif par $Type^{\#}$. L'ensemble $Type^{\#}$ est supposé fini (et de taille relativement restreinte) et muni d'un ordre partiel noté $\leq$. Cet ensemble ordonné admet un minimum noté $\bot$ et un maximum noté $\top$.

On suppose également que l'on dispose d'un opérateur de borne supérieure noté $\sqcup$.

Finalement, il existe soit une fonction de concrétisation

$$\gamma : Type^{\#} \longrightarrow \wp(Type)$$

monotone relativement à $\leq$ et à l'inclusion; soit une fonction d'abstraction

$$\alpha : Type \longrightarrow Type^{\#}.$$

Donnons maintenant deux exemples de domaines primitifs.

4.1.1 Exemple 1 : ensembles de types

On peut définir $Type^{\#}$ comme l'ensemble des parties de l'ensemble $Type$. L'ordre partiel se réduit alors à l'inclusion et la borne supérieure à l'union. La fonction de concrétisation est alors simplement l'identité.

Quant à la fonction d'abstraction, elle se définit par

$$\begin{array}{rccl} \alpha : & Type & \longrightarrow & \wp(Type) = Type^{\#} \\ & t & \rightsquigarrow & \{t\}. \end{array}$$

L'élément minimum $\bot$ est bien sûr l'ensemble vide et l'élément maximal $\top$ l'ensemble $Type$.

4.1.2 Exemple 2 : types

On peut également définir $Type^{\#}$ comme l'ensemble des types lui-même. Il faut cependant lui adjoindre un élément maximum supplémentaire :

$$Type^{\#} = Type + \{\top\}.$$

L'ordre partiel est alors fourni par la relation de spécialisation $\preceq_{\pi}$ étendue à $\top$.

La fonction de concrétisation se définit comme suit.

$$\begin{array}{rccl} \gamma: & Type + \{\top\} & \longrightarrow & \wp(Type) \\ & t & \rightsquigarrow & \{t' \mid t' \preceq_\pi t\} \\ & \top & \rightsquigarrow & Type \end{array}$$

La fonction d'abstraction se réduit à l'identité. La borne supérieure de deux types t et t' est le plus petit type t'' tel que t et t' héritent de t'' si un tel type existe ou $\top$ si ce n'est pas le cas.

Ici, **bot** joue le rôle de l'élément minimum.

4.2 Forme du domaine

Dans cette section, on définit le domaine d'interprétation abstrait. Pour fixer les idées, voyons les éléments de celui-ci selon l'optique "concrétisation", i.e. voyons chaque élément du domaine abstrait comme une représentation d'un ensemble d'éléments du domaine concret.

Comme on l'a mentionné, on opte pour des éléments abstraits "homorphiques" aux éléments concrets. Cela signifie qu'on tente de préserver au maximum la structure et les propriétés propres au domaine concret $\mathbb{D}^*$.

Intuitivement, abstraire une valeur de base revient simplement à ne conserver que le type de celle-ci. Mais que penser des instances ? Le domaine proposé ici repose sur une "vue binaire" des choses : soit on connaît encore tout sur l'instance (i.e. on connaît non seulement l'instance elle-même mais aussi toutes ses instances supérieures), soit tout ce qu'on sait sur cette instance est une approximation du type de celle-ci.

Une autre caractéristique du domaine réside dans le traitement du "sharing" : en effet, que signifie un partage de structure au niveau abstrait ? Nous décidons de voir le "sharing" comme une seconde source de perte d'information, c'est-à-dire que si un élément du domaine abstrait présente un point de partage, cet élément représente des éléments concrets avec ou sans le dit point de partage.

Passons maintenant à la définition de ce domaine. Le domaine étant "homomorphique" au domaine concret, il semble opportun de reprendre le même schéma de définition que pour ce dernier: nous commençons donc par préciser quel est l'ensemble des valeurs concrètes; ensuite, nous présentons les notions d'environnement et de store abstrait pour en dériver finalement la définition du domaine abstrait.

4.2.1 Valeurs abstraites

On distingue trois catégories de valeurs abstraites : les types, les instances abstraites et les éléments du domaine abstrait primitif. Les éléments de $Type$ correspondent à des valeurs "de base" du domaine concret pour lesquelles, au niveau du type, on n'a perdu aucune information.

Les instances abstraites correspondent aux instances concrètes pour lesquelles on a conservé une information de type complète. Les éléments du domaine abstrait primitif peuvent aussi bien correspondre à des valeurs concrètes "de base" qu'à des instances concrètes; la seule information conservée étant alors une approximation du type de l'objet.

Définition 4.1 (Valeurs abstraites)

L'ensemble des valeurs abstraites est noté $\backslash Val^{\#}$. *Il est défini comme l'union disjointe*

$$\backslash Val^{\#} = Type + Type^{\#} + \mathbb{I}nst^{\#}$$

où $\mathbb{I}nst^{\#}$ *désigne l'ensemble des instances abstraites.*

$$\mathbb{I}nst^{\#} \subseteq Nclasse \times (Nchamp + \{\mathbf{super}\} \nrightarrow \mathbb{L}oc^{\#})$$

Si (nc, v) *appartient à* $\mathbb{I}nst^{\#}$ *alors* v *est injective.*

L'ensemble $\mathbb{L}oc^{\#}$ rassemble les locations abstraites.

4.2.2 Environnements et stores abstraits

L'environnement et le store se calquent sur leurs homologues concrets : l'environnement fournit la location associée à une variable ou au désignateur **this** ainsi que le type statique de ces éléments; le store associe aux différentes locations abstraites une valeur abstraite.

Définition 4.2 (Environnements et stores abstraits)

$\mathbb{E}nv^{\#}$ *désigne l'ensemble des environnements abstraits.*

$$\mathbb{E}nv^{\#} \subseteq Nvar + \{\mathbf{this}\} \nrightarrow Type \times \mathbb{L}oc^{\#}$$

Si e *appartient à* $\mathbb{E}nv^{\#}$ *et si* x *et* y *sont des éléments distincts du domaine de* e, *alors*

$$p_2(e(x)) \neq p_2(e(y)).$$

$\$tore^{\#}$ *désigne l'ensemble des stores abstraits.*

$$\$tore^{\#} = \mathbb{L}oc^{\#} \nrightarrow \backslash Val^{\#}$$

On note[1] $\perp_0^{\#}$, l'élément de $\mathbb{E}nv^{\#}$ de domaine vide et $\perp_1^{\#}$ l'élément de $\$tore^{\#}$ de domaine vide.

Tout comme au niveau concret, pour définir notre domaine d'interprétation, il nous reste maintenant à énoncer les propriétés souhaitables d'un élément du produit cartésien $\mathbb{E}nv^{\#} \times \$tore^{\#}$.

[1] On ometra d'apposer un "#" en exposant dans les contextes ne permettant pas de confusion avec les éléments concrets.

4.2.3 Domaine abstrait

Vu l'homomorphisme reliant les domaines concrets et abstraits, il semble normal, qu'à leur tour, les propriétés définissant le domaine d'interprétation imitent leurs homologues concrètes.

Soit donc (a_0, a_1) un élément du produit cartésien $\mathbb{E}nv^\# \times \$tore^\#$. Quelles sont les propriétés que doit vérifier ce couple pour être un élément du domaine ?

Propriété 1 : Forme de l'environnement

$$a_0 = \bot_0[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$$

avec

$$n \in \mathbb{N},\ t_1, ..., t_n \neq \mathbf{bot},\ t_{n+1} \in Nclasse.$$

Propriété 2 : Unicité des locations

$$a_1(l) = (nc, v) \wedge a_1(l') = (nc', v') \Rightarrow (a_1(l) = a_1(l') \vee Im(v) \cap Im(v') = \emptyset).$$

Propriété 3 : Coupure du store

Si on définit la suite de niveaux (A_j) de limite A par

$$A_0 = \{\alpha_i \mid 1 \leq i \leq n+1\}$$

$$A_{j+1} = A_j \cup \bigcup_{\substack{l \in A_j \\ a_1(l) \in \mathbb{I}nst^\#}} Im(p_2(a_1(l)))$$

on doit avoir l'égalité

$$A = dom(a_1).$$

Propriété 4 : Respect des types statiques

Si $a_1(l) = (nc, v)$ est un élément de $\mathbb{I}nst^\#$, le store doit vérifier les conditions ci-après.

- $\mathbf{super} \in dom(v) \Leftrightarrow nc \in dom(\pi)$
- $\forall ch \in Nchamp, ch \in dom(v) \Leftrightarrow ch \in dom(\epsilon^0_{ch} nc)$
- $\mathbf{super} \in dom(v) \Rightarrow a_1(v(\mathbf{super}))) = (\pi(nc), p) \in \mathbb{I}nst^\#$
- $ch \in dom(v) \Rightarrow \begin{cases} \mathbf{soit} & a_1(v(ch)) \in Type \ \wedge\ a_1(v(c)) \preceq_\pi \epsilon^0_{ch}\ nc\ ch \\ \mathbf{soit} & a_1(v(ch)) = e \in Type^\# \ \wedge\ \exists t \in \gamma(e), t \preceq_\pi \epsilon^0_{ch}\ nc\ ch \\ \mathbf{soit} & a_1(v(ch)) \in \mathbb{I}nst^\# \ \wedge\ p_1(a_1(v(ch)) \preceq_\pi \epsilon^0_{ch}\ nc\ ch \end{cases}$

Soulignons la différence de traitement entre le pseudo-champ **super** et les autres champs. Dans ce domaine, soit on ne connaît plus d'une instance qu'une approximation de son type soit on connaît la structure complète de celle-ci : cela signifie que si le store fournit comme image d'une location une instance abstraite dont la classe étend une autre classe, le store doit associer à la location attachée au pseudo-champ **super** de cette instance une autre instance. Il n'en va naturellement pas de même pour les autres champs qui peuvent être à leur tour "abstraits".

Dans les cas où le store associe à un champ une valeur abstraite primitive e, on a envie de requérir que tous les types concrets représentés par e soient des spécialisations du type statique du champ. Il se peut cependant que l'on ait un partage avec un autre élément d'un type "différent" (par exemple un type "plus grand"). Tout ce qu'on peut exiger est, par conséquent, l'existence d'au moins une spécialisation du type statique du champ dans la concrétisation de e.

L'information renvoyée par le store pour les champs peut aussi être un type concret mais uniquement dans deux cas : soit il s'agit d'un "champ mis à **null**" et le type est alors **bot**, soit il s'agit d'un champ de type entier ou booléen (en effet, les autres cas sont impossibles car un champ est toujours initialisé).

Passons maintenant au respect des types mentionnés dans l'environnement. Pour les variables locales, on exprime que le "type dynamique" doit être une spécialisation du type statique tandis que, pour **this**, le "type dynamique" doit correspondre "exactement" au type statique. Vu la structure du domaine, on a chaque fois trois cas disjoints.

$$1 \leq i \leq n \Rightarrow \begin{cases} \textbf{soit} & a_1(\alpha_i)) = t \in \mathit{Type} \ \wedge \ t \preceq_\pi t_i \\ \textbf{soit} & a_1(\alpha_i) = e \in \mathit{Type}^\# \ \wedge \ \exists t \in \gamma(e), t \preceq_\pi t_i \\ \textbf{soit} & a_1(\alpha_i) \in \mathit{Inst}^\# \ \wedge \ p_1(a_1(\alpha_i)) \preceq_\pi t_i \end{cases}$$

$$\begin{cases} \textbf{soit} & a_1(\alpha_{n+1})) = t \in \mathit{Type} \ \wedge \ t = t_{n+1} \\ \textbf{soit} & a_1(\alpha_{n+1}) = e \in \mathit{Type}^\# \ \wedge \ t_{n+1} \in \gamma(e) \\ \textbf{soit} & a_1(\alpha_{n+1}) \in \mathit{Inst}^\# \ \wedge \ p_1(a_1(\alpha_{n+1})) = t_{n+1} \end{cases}$$

Définition 4.3 (Domaine d'interprétation abstrait)

Le domaine d'interprétation abstrait est noté $\mathbb{D}^\#$ et est défini par la relation d'appartenance suivante : a appartient à $\mathbb{D}^\#$ si et seulement si a est un élément du produit $\mathbb{E}nv^\# \times \$tore^\#$ qui vérifie les propriétés 1, 2, 3 et 4.

4.3 Fonction de concrétisation

Pour réellement définir un domaine abstrait, il faut "donner un sens" à ses éléments, c'est-à-dire qu'il faut définir une relation précise entre les éléments du domaine abstrait et ceux du domaine concret.

Pour parvenir à cela, une première idée est de caractériser précisément l'ensemble des points

du domaine concret représentés par un point du domaine abstrait. On définit alors ce que l'on dénomme couramment une fonction de concrétisation. Cette fonction fournit en quelque sorte la "sémantique" des élements du domaine abstrait. Il est évident que la définition de cette fonction de concrétisation s'appuiera sur la fonction de concrétisation du domaine primitif.

Tout au long de la présentation de ce domaine, nous avons insisté sur le lien "homomorphique" existant entre élements abstraits et éléments concrets. Il ne serait donc pas étonnant de relier ceux-ci par un homomorphisme au sens mathématique du terme et, en effet, c'est par le biais de fonctions respectant certaines propriétés des espaces en jeu que nous définissons notre fonction de concrétisation.

Soient $a = (a_0, a_1)$ un élément du produit cartésien $\mathbb{E}nv^{\#} \times \$tore^{\#}$, $d = (d_0, d_1)$ un élément de $\mathbb{E}nv \times \$tore$ et f une fonction des locations concrètes vers les locations abstraites.

Nous introduisons la notation $d \xrightarrow{f} a$ qui se lit "a approxime d au travers de f", ce qui signifie grosso-modo que f "respecte" la structure, les propriétés de d.

Définition 4.4 $(d \xrightarrow{f} a)$

Soient a appartenant à $\mathbb{E}nv^{\#} \times \$tore^{\#}$ et d appartenant à $\mathbb{E}nv \times \$tore$.
On dira que a approxime d au travers de f, i.e. $d \xrightarrow{f} a$, si et seulement si a, d et f vérifient les trois propriétés ci-dessous.

1. $f : dom(d_1) \rightarrowtail dom(a_1)$

2. $d_0 = \perp_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \mathbf{this}/t_{n+1}\delta_{n+1}]$
   $a_0 = \perp_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $\forall i : 1 \leq i \leq n+1 : f(\delta_i) = \alpha_i$

3. $\forall l \in dom(f)$,

$$a_1(f(l)) = t \in Type \Rightarrow d_1(l) \in \mathbb{B}ase \wedge (p_1(d_1(l)) = t$$
$$a_1(f(l)) = e \in Type^{\#} \Rightarrow p_1(d_1(l)) \in \gamma(e)$$
$$a_1(f(l)) = (nc, va) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} d_1(l) = (nc, v) \in \mathbb{I}nst \\ dom(v) = dom(va) \\ \forall ch \in dom(v),\ va(ch) = f(v(ch)) \end{cases}$$

Intuitivement, pour qu'un élément abstrait approxime un élément concret, il faut d'abord que l'environnement du premier puisse s'ajuster sur l'environnement du second. Ensuite, il faut que, si en point du domaine du store abstrait on rencontre une instance abstraite, "aux points correspondants" du store concret on trouve des instances concrètes "équivalentes"; de même, lorsqu'on rencontre un type.

En revanche, si le store abstrait associe à un point du domaine un élément de $Type^{\#}$, le store concret peut associer "aux points correpondants" aussi bien des valeurs de base que des instances, pour peu bien sûr que l'élément de $Type^{\#}$ soit une approximation correcte pour ceux-ci.

C'est cette notion de "points correspondants" que traduit la fonction f. Une des caractéristiques de notre domaine réside par conséquent dans l'aspect fonctionnel de cette mise en correspondance : deux locations abstraites différentes ne peuvent représenter la même location concrète. Inversément, on n'exige absolument pas l'injectivité de f : une même location abstraite peut représenter plusieurs locations concrètes.

Ces propriétés expriment en fait le choix mentionné précédemment pour le "sharing" : le "sharing" au niveau abstrait est une perte d'information qui se traduit éventuellement par l'amalgame de plusieurs locations concrètes.

L'aspect partiel de cette fonction de mise en correspondance résulte du fait que lorsqu'on abstrait un objet, on perd l'information concernant la structure de celui-ci. Il semble donc assez logique que les locations soutenant cette structure n'aient pas nécessairement de contreparties abstraites.

Nous avons parlé de l'aspect partiel, fonctionnel et non injectif de cette mise en correspondance; qu'en est-il de la surjectivité ? Intuitivement, il paraît tout à fait non naturel d'avoir des locations abstraites ne correpondant à aucune location concrète. Pourquoi alors ne pas exiger la surjectivité de la fonction f? La réponse à cette question est fournie par la proposition 4.2.

Proposition 4.1 (Respect des niveaux du store)

Soit d un élément de $\mathbb{E}nv \times \mathbb{S}tore$.
Désignons par (D_j) la suite de niveaux attachée au domaine de d_1.
Soit également a appartenant au produit $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
On désigne par (A_j) la suite de niveaux attachée au domaine de a_1.
On a alors l'implication ci-dessous.

$$d \xrightarrow{f} a \Rightarrow \forall i,\ A_i \subseteq f(D_i).$$

Preuve :

On procède par récurrence sur l'indice du niveau. On prouve donc les deux assertions intermédiaires suivantes.

1. $A_0 \subseteq f(D_0)$
2. $A_i \subseteq f(D_i) \Rightarrow A_{i+1} \subseteq f(D_{i+1})$

La première assertion est triviale puisque, d'une part, $A_0 = \{\alpha_1, ..., \alpha_{n+1}\}$ et $D_0 = \{\delta_1, ..., \delta_{n+1}\}$ et, d'autre part,

$$\forall i : 1 \leq i \leq n+1 : f(\delta_i) = \alpha_i.$$

Passons maintenant à l'étape de récurrence.

La définition de la suite (A_j) nous donne une expression de A_{i+1} en fonction de A_i :

$$A_{i+1} = A_i \cup \bigcup_{\substack{l \in A_i \\ a_1(l) \in \mathbb{I}nst^\#}} Im(p_2(a_1(l))).$$

En appliquant deux fois l'hypothèse de récurrence, cette égalité fournit l'inclusion ci-après.

$$A_{i+1} \subseteq f(D_i) \cup \bigcup_{\substack{l \in f(D_i) \\ a_1(l) \in \mathbb{I}nst^\#}} Im(p_2(a_1(l)))$$

Attendu la définition de l'image d'un ensemble, cette inclusion se réécrit

$$A_{i+1} \subseteq f(D_i) \cup \bigcup_{\substack{l \in D_i \cap dom(f) \\ a_1(f(l)) \in \mathbb{I}nst^\#}} Im(p_2(a_1(f(l)))).$$

Comme $d \xrightarrow{f} a$, on obtient l'inclusion

$$A_{i+1} \subseteq f(D_i) \cup \bigcup_{\substack{l \in D_i \cap dom(f) \\ a_1(f(l)) \in \mathbb{I}nst^\#}} f(Im(p_2(d_1(l)))).$$

Pour que l'on ait une instance au niveau abstrait, il faut qu'on ait une instance au niveau concret. Par conséquent, on peut élargir l'inclusion ci-dessus comme suit.

$$A_{i+1} \subseteq f(D_i) \cup \bigcup_{\substack{l \in D_i \cap dom(f) \\ d_1(l) \in \mathbb{I}nst}} f(Im(p_2(d_1(l)))) \tag{4.1}$$

Remarquons finalement que le membre de droite de cette dernière inclusion n'est autre que $f(D_{i+1})$ puisque D_{i+1} est défini par

$$D_{i+1} = D_i \cup \bigcup_{\substack{l \in D_i \\ d_1(l) \in \mathbb{I}nst}} Im(p_2(d_1(l))).$$

$\Diamond$

Soulignons que l'on a pas l'inclusion inverse : toutes les transformations d'ensembles évoquées au cours de la preuve ci-avant pourraient en fait conserver des égalités, sauf la transformation utilisée pour aboutir à l'inclusion (4.1).

Proposition 4.2 (Surjectivité de la fonction de mise en correspondance)

Soient d appartenant à $\mathbb{D}^*$ et a appartenant à $\mathbb{E}nv^\# \times \mathbb{S}tore^\#$.
Si $d \xrightarrow{f} a$, on a l'implication : $a \in \mathbb{D}^\# \Rightarrow f$ surjective.

Preuve :

Cette preuve découle quasi directement de la proposition précédente.

Par définition des domaines concret et abstrait, on a les deux implications suivantes.

$d \in \mathbb{D}^* \Rightarrow dom(d_1) = D$
$a \in \mathbb{D}^\# \Rightarrow dom(a_1) = A$

Les deux suites de niveaux stabilisent à un moment donné. Soit n un indice postérieur à la stabilisation des deux suites. On peut alors écrire

$$dom(a_1) = A = A_n \subseteq f(D_n) = f(D) = f(dom(d_1)).$$

Cette inclusion exprime clairement que f ne peut être que surjective. $\Diamond$

On aimerait écrire une propriété nous disant que si a approxime un élément d de $\mathbb{D}^*$ au travers d'une fonction surjective f, a est nécessairement un élément de $\mathbb{D}^\#$. Pour parvenir à cela, il nous faudrait alourdir la définition de l'approximation. En effet, notre définition ne garantit pas, par exemple, la conservation de l'unicité des locations. Elle ne garantit pas non plus la conservation de la structure "complète" des instances.

Revenons maintenant à la définition de la fonction de concrétisation : un élément du domaine abstrait représente simplement l'ensemble des éléments du domaine concret pouvant être mis en correspondance avec celui-ci.

Définition 4.5 (Fonction de concrétisation)

La fonction de concrétisation se note γ^* et est définie comme suit.

$$\begin{array}{rccl} \gamma^* : & \mathbb{D}^\# & \longrightarrow & \wp(\mathbb{D}^*) \\ & a & \rightsquigarrow & \{d \mid \exists f,\ d \xrightarrow{f} a\} \end{array}$$

4.4 Fonction d'abstraction

Pour définir les liens entre éléments du domaine concret et éléments du domaine abstrait, on peut suivre l'idée "inverse" à celle présentée au cours de la section précédente. Au lieu de fournir une *fonction de concrétisation*, on fournit une *fonction d'abstraction* : cette fonction donne pour chaque élément concret sa "meilleure" représentation abstraite.

Le travail d'une telle fonction se réduit ici à "sectionner" les informations non relatives aux types des objets. Dans cette optique, les "difficultés" émergent lorsqu'il faut "mettre ensemble" plusieurs éléments abstraits.

Pour pouvoir définir l'abstraction d'un élément de $\mathbb{D}^*$, on pose l'hypothèse qu'il existe une

injection canonique des locations concrètes dans les locations abstraites, notée in.

$$in : \mathbb{L}oc \longmapsto \mathbb{L}oc^{\#}$$

Définition 4.6 (Fonction d'abstraction)

La fonction d'abstraction possède la signature

$$\alpha^* : \mathbb{D}^* \longrightarrow \mathbb{E}nv^{\#} \times \$tore^{\#}$$

et si d représente un élement de $\mathbb{D}^*$ tel que

$$d_0 = \bot_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \mathbf{this}/t_{n+1}\delta_{n+1}].$$

son image, notée a, par la fonction α^* se construit comme suit.

1. $a_0 = \bot_0^{\#}[v_1/t_1 in(\delta_1), ..., v_n/t_n in(\delta_n), \mathbf{this}/t_{n+1} in(\delta_{n+1})]$
2. $dom(a_1) = in(dom(d_1))$
3. $\forall l \in dom(d_1)$,

$$d_1(l) \in \mathbb{B}ase \Rightarrow a_1(in(l)) = p_1(d_1(l))$$

$$d_1(l) = (nc, v) \in \mathbb{I}nst \Rightarrow \begin{cases} a_1(in(l)) = (nc, va) \in \mathbb{I}nst^{\#} \\ dom(v) = dom(va) \\ \forall ch \in dom(v),\ va(ch) = in(v(ch)) \end{cases}$$

Soulignons que cette fonction définit l'abstraction d'un seul élément concret et n'est donc la fonction d'abstraction intervenant dans une "insertion de Galois".

Proposition 4.3 (Appartenance au domaine)

Si d appartient à $\mathbb{D}^*$, alors $\alpha^*(d)$ appartient à $\mathbb{D}^{\#}$.

Preuve :

Cette propriété est triviale car il est évident que α^* ne modifie absolument pas la structure de d (vu que in est injective). $\diamondsuit$

4.5 Equivalence intuitive

La dépendance de la fonction d'abstraction par rapport à l'injection canonique rappelle que le choix des locations est dans une certaine mesure arbitraire.

Cette constation suggère d'introduire une notion d'équivalence sur $\mathbb{D}^{\#}$: deux éléments de $\mathbb{D}^{\#}$ sont équivalents s'il sont identiques "au choix" des locations près, i.e. s'ils sont isomorphes. Il s'agit donc d'une relation d'équivalence tout à fait intuitive qui se traduit naturellement par l'existence d'une bijection entre les locations des deux éléments en relation.

Définition 4.7 (Equivalence intuitive)

Soient deux éléments a et a' de $\mathbb{D}^{\#}$. Ces deux éléments sont dits intuitivement équivalents (ce qu'on note $a \equiv a'$) si et seulement si il existe une fonction f qui respecte les trois conditions suivantes.

1. $f : dom(a_1) \rightarrowtail\!\!\!\!\!\rightarrow dom(a'_1)$

2. $a_0 = \bot_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $a'_0 = \bot_0^{\#}[v_1/t_1\alpha'_1, ..., v_n/t_n\alpha'_n, \mathbf{this}/t_{n+1}\alpha'_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : f(\alpha_i) = \alpha'_i$

3. $\forall l \in dom(a_1)$,

$$
\begin{array}{l}
a'_1(f(l)) = t \in Type \Leftrightarrow a_1(l) = t \\
a'_1(f(l)) = e \in Type^{\#} \Leftrightarrow a_1(l) = e \\
a'_1(f(l)) = (nc, v')\mathbb{I}nst^{\#} \Leftrightarrow a_1(l) = (nc, v) \in \mathbb{I}nst \\
a'_1(f(l)) = (nc, v') \in \mathbb{I}nst^{\#} \wedge a_1(l) = (nc, v) \in \mathbb{I}nst \\
\qquad \Rightarrow \left\{ \begin{array}{l} dom(v) = dom(v') \\ \forall ch \in dom(v'),\ v'(ch) = f(v(ch)) \end{array} \right.
\end{array}
$$

Proposition 4.4 (Relation d'équivalence)

La relation $\equiv$ de la définition 4.7 est effectivement une relation d'équivalence sur $\mathbb{D}^{\#}$, i.e. elle est réflexive, transitive et symétrique.

Preuve :

Pour prouver la réflexivité, il suffit de constater que l'identité vérifie les trois conditions demandées.

La transitivité est également triviale puisque la composée de deux bijections est une bijection.

Bien que la preuve de la symétrie ne soit pas plus compliquée, attardons-nous quelque peu sur celle-ci.

Puisque f est une bijection, f admet une réciproque $f^{\leftarrow}$ qui est également une bijection. On a donc bien la première condition :

$f^{\leftarrow} : dom(a'_1) \rightarrowtail\!\!\!\!\!\rightarrow dom(a_1)$.

La deuxième condition est évidemment vérifiée puisque

$$\forall i : 1 \leq i \leq n+1 : f^{\leftarrow}(\alpha'_i) = f^{\leftarrow}(f(\alpha_i)) = \alpha_i.$$

Passons donc à la troisième condition.

Soit $l \in dom(a'_1)$. Comme f est une bijection, il existe une et une seule location ll telle que $f(ll) = l$.

$a_1(f^{\leftarrow}(l)) = t \in Type$ se réécrit par conséquent $a_1(ll) = t \in Type$.

Par hypothèse, cette égalité équivaut à l'égalité $a'_1(f(ll)) = t$, qui se réécrit à son tour

$$a'_1(f(f^{\leftarrow}(l))) = t,$$

c'est-à-dire $a'_1(l) = t$.

Les deux autres cas sont parfaitement analogues. $\Diamond$

4.6 Préordre

Comme on l'a signalé, on désire s'intéresser à la *borne supérieure* de plusieurs éléments du domaine abstrait. Cet intérêt résulte, d'une part, du sentiment qu'une opération de majoration de plusieurs éléments du domaine abstrait risque d'être essentielle pour la constrution des règles de la sémantique abstraite et, d'autre part, du fait qu'usuellement le lien entre *concrétisation* et *abstraction* se réalise via ce genre d'opérateurs.

Mais toute notion de borne supérieure ou de majorant sous-entend la présence d'une relation d'ordre.

La définition de la relation d'équivalence précédente nous pousse cependant à penser qu'en fait, vu l'aspect arbitraire des locations, nous devrons ici nous contenter d'une relation de préordre (à moins évidemment de passer aux classes d'équivalence).

On peut donc déjà, a priori, éliminer toute notion de borne supérieure unique. L'unicité de la borne supérieure résulte en effet de la propriété d'anti-symétrie de la relation d'ordre sous-jacente. On souhaite cependant pouvoir obtenir une borne supérieure unique "à un isomorphisme près".

Passons maintenant à la définition d'une relation de préordre sur $\mathbb{D}^{\#}$. Intuitivement, un élément a' de $\mathbb{D}^{\#}$ est plus grand qu'un autre élément a de $\mathbb{D}^{\#}$ s'il "approxime" celui-ci, i.e. si l'information fournie par a' est moins précise que l'information fournie par a mais cohérente avec celle-ci. Dans notre cas, il s'agit donc d'une part "d'approximer" les informations de type et d'autre part de conserver les "points de partage".

Il semble assez naturel de proposer, pour cette relation de préordre, une définition similaire à celle proposée pour la fonction de concrétisation. Cette définition s'appuie évidemment sur la relation d'ordre du domaine primitif.

Définition 4.8 ($a \xrightarrow{f} a'$)

Soient a et a' appartenant à $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
On dira que a' approxime a au travers de f, i.e. $a \xrightarrow{f} a'$, si et seulement si a, a' et f vérifient les trois propriétés ci-dessous.

1. $f : dom(a_1) \rightarrowtail dom(a'_1)$

2. $a_0 = \bot_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $a'_0 = \bot_0^{\#}[v_1/t_1\alpha'_1, ..., v_n/t_n\alpha'_n, \mathbf{this}/t_{n+1}\alpha'_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : f(\alpha_i) = \alpha'_i$

3. $\forall l \in dom(f)$,

$$a'_1(f(l)) = t \in Type \Rightarrow a_1(l) = t$$

$$a'_1(f(l)) = e \in Type^{\#} \Rightarrow \begin{cases} \textbf{soit } a_1(l) \in Type \wedge a_1(l) \in \gamma(e) \\ \textbf{soit } a_1(l) \in Type^{\#} \wedge a_1(l) \leq e \\ \textbf{soit } a_1(l) = (nc, v) \in \mathbb{I}nst^{\#} \wedge nc \in \gamma(e) \end{cases}$$

$$a'_1(f(l)) = (nc, v') \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} a_1(l) = (nc, v) \in \mathbb{I}nst^{\#} \\ dom(v) = dom(v') \\ \forall ch \in dom(v),\ v'(ch) = f(v(ch)) \end{cases}$$

Cette définition utilise la fonction de concrétisation du domaine primitif. Soulignons, que moyennant quelques très légères modifications, on pourrait tout aussi bien se baser sur la fonction d'abstraction de $Type^{\#}$.

La figure 4.1 donne un exemple d'approximations successives : la première situation fournit une information "complète" tant au niveau des types qu'au niveau du "sharing"; cette première situation est approximée au travers de la fonction f par la deuxième situation qui conserve une information complète au niveau des types mais ne permet plus d'affirmer que les deux variables désignent des instances distinctes; cette deuxième situation est elle-même approximée par la troisième situation au travers de la fonction f'; dans cette dernière situation, on sait uniquement que x et y sont de type A.

On retrouve des propriétés tout à fait analogues à celles énoncées au niveau de la définition de la fonction de concrétisation.

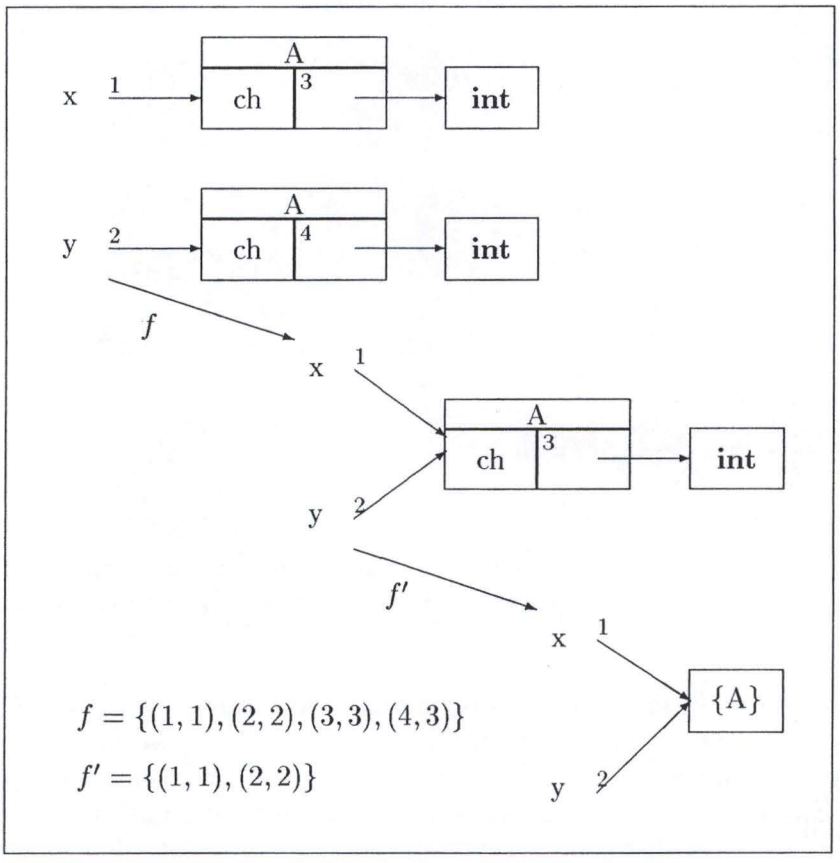

Figure 4.1: Exemples d'approximations

Proposition 4.5 (Respect des niveaux du store)

Soit a un élément de $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
Désignons par (A_j) la suite de niveaux attachée au domaine de a_1.
Soit également a' appartenant au produit $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
On désigne par (A'_j) la suite de niveaux attachée au domaine de a'_1.
On a alors l'implication ci-dessous.

$$a \xrightarrow{f} a' \Rightarrow \forall i,\ A'_i \subseteq f(A_i).$$

Preuve :

La preuve de cette proposition est tout à fait similaire à celle fournie pour la proposition 4.1.$\Diamond$

Naturellement, on a toujours pas l'inclusion inverse.

Proposition 4.6 (Surjectivité de la fonction de mise en correspondance)

Soit a appartenant à $\mathbb{D}^{\#}$ et a' appartenant à $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
Si $a \xrightarrow{f} a'$, on a l'implication : $a' \in \mathbb{D}^{\#} \Rightarrow f$ surjective.

La définition du préordre découle directement de la définition 4.8.

Définition 4.9 (Préordre sur $\mathbb{D}^{\#}$)

Soient deux éléments a et a' de $\mathbb{D}^{\#}$. On dira que a' approxime a (ce qu'on note $a \leq^* a'$) si et seulement si il existe une fonction f telle que $a \xrightarrow{f} a'$.

Proposition 4.7 (Relation de préordre)

La relation $\leq^*$ définie sur $\mathbb{D}^{\#}$ est effectivement une relation de préordre, i.e. elle est réflexive et transitive.

Preuve :

Il est évident qu'on a bien $a \xrightarrow{id} a$.

On prouve facilement, mais fastidieusement, $a \xrightarrow{g \circ f} c$ à partir de $a \xrightarrow{f} b$ et $b \xrightarrow{g} c$ (une preuve analogue est fournie pour la proposition suivante). $\Diamond$

On clôt cette section par une propriété essentielle de cohérence entre la fonction de concrétisation et le préordre défini sur $\mathbb{D}^{\#}$: il faut que si a' approxime a, a' représente tous les éléments du domaine concret représentés par a.

Proposition 4.8 (Cohérence de $\leq^*$ par rapport à γ^*)

Soient a et a' appartenant à $\mathbb{D}^\#$. Si a' approxime a, la concrétisation de a' contient la concrétisation de a, i.e.

$$a \leq^* a' \Rightarrow \gamma^*(a) \subseteq \gamma^*(a').$$

Preuve :

Soit $d \in \gamma^*(a)$. Montrons que $d \in \gamma^*(a')$.

Puisque $d \in \gamma^*(a)$, il existe une fonction f telle que $d \xrightarrow{f} a$. D'autre part, puisque $a \leq^* a'$, il existe une fonction g telle que $a \xrightarrow{g} a'$.

Si la composée de g et f vérifie $d \xrightarrow{g \circ f} a'$, on a bien que d est un élément de $\gamma^*(a')$.

1. $g \circ f : dom(d_1) \rightarrowtail dom(a'_1)$
2. $d_0 = \bot_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \mathbf{this}/t_{n+1}\delta_{n+1}]$
   $a_0 = \bot_0^\#[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $a'_0 = \bot_0^\#[v_1/t_1\alpha'_1, ..., v_n/t_n\alpha'_n, \mathbf{this}/t_{n+1}\alpha'_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : g \circ f(\delta_i) = g(f(\delta_i)) = g(\alpha_i) = \alpha'_i$
3. Soit $l \in dom(g \circ f)$.

   Par définition de la composée de fonctions partielles, $l \in dom(f)$ et $f(l) \in dom(g)$.

   $a'(g(f(l)) = t \in Type \Rightarrow a(f(l)) = t \Rightarrow d(l) = t$

   La première implication découle du fait que $a \xrightarrow{g} a'$ et la seconde du fait que $d \xrightarrow{f} a$.

   De même, puisque $a \xrightarrow{g} a'$,

$$a'_1(g(f(l))) = e \in Type^\# \Rightarrow \begin{cases} \textbf{soit } a_1(f(l)) \in Type \wedge a_1(f(l)) \in \gamma(e) \\ \textbf{soit } a_1(f(l)) \in Type^\# \wedge a_1(f(l)) \leq e \\ \textbf{soit } a_1(f(l)) = (nc, va) \in \mathbb{I}nst^\# \wedge nc \in \gamma(e) \end{cases}$$

   et puisque $d \xrightarrow{f} a$,

   $a_1(f(l)) \in Type \Rightarrow d_1(l) \in \mathbb{B}ase \wedge p_1(d_1(l)) = a_1(f(l))$
   $a_1(f(l)) \in Type^\# \Rightarrow p_1(d_1(l)) \in \gamma(a_1(f(l)))$

$$a_1(f(l)) = (nc, va) \in \mathbb{I}nst^\# \Rightarrow \begin{cases} d_1(l) = (nc, v) \in \mathbb{I}nst \\ dom(v) = dom(va) \\ \forall ch \in dom(v), va(ch) = f(v(ch)) \end{cases} \quad (4.2)$$

On obtient donc

$$a'_1(g(f(l))) = e \in Type^{\#} \Rightarrow \begin{cases} \textbf{soit } d_1(l) \in \mathbb{B}ase \wedge p_1(d_1(l)) = a_1(f(l)) \in \gamma(e) \\ \textbf{soit } a_1(f(l)) \in Type^{\#} \wedge p_1(d_1(l)) \in \gamma(a_1(f(l))) \subseteq \gamma(e) \\ \qquad \text{(puisque } \gamma \text{ monotone par rapport à } \leq) \\ \textbf{soit } d_1(l) = (nc, v) \in \mathbb{I}nst \wedge nc \in \gamma(e) \end{cases}$$

En bref, on a donc ce qu'on souhaite :
$a'_1(g(f(l))) = e \in Type^{\#} \Rightarrow p_1(d_1(l)) \in \gamma(e)$

Passons enfin au dernier cas de figure.

$$a'_1(g(f(l))) = (nc, va') \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} a_1(f(l)) = (nc, va) \in \mathbb{I}nst^{\#} \\ dom(va) = dom(va') \\ \forall ch \in dom(va),\ va'(ch) = g(va(ch)) \end{cases}$$

Combinée avec l'implication (4.2), l'implication ci-dessus fournit :

$$a'_1(g(f(l))) = (nc, va') \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} d_1(l) = (nc, v) \in \mathbb{I}nst \\ dom(va) = dom(va') = dom(v) \\ \forall ch \in dom(v),\ va'(ch) = g(va(ch)) = g(f(v(ch)) \end{cases}$$

Et la troisième condition est vérifiée.

◊

4.7 Equivalence classique

Nous disposons maintenant d'un préordre sur $\mathbb{D}^{\#}$. On peut trivialement déduire de ce préordre une seconde relation d'équivalence que nous nommerons *équivalence classique.*

Définition 4.10 (Equivalence classique)

Soient deux éléments a et a' de $\mathbb{D}^{\#}$. Ces deux éléments sont dits classiquement équivalents (ce qu'on note $a \approx a'$) si et seulement si $a \leq^* a'$ et $a' \leq^* a$.

Proposition 4.9 (Relation d'équivalence)

La relation $\approx$ définie sur $\mathbb{D}^{\#}$ est effectivement une relation d'équivalence, i.e elle est réflexive, symétrique et transitive.

Preuve :

Il s'agit effectivement d'une relation réflexive et transitive puisque $\leq^*$ est elle même réflexive et transitive. La symétrie est évidente vu la forme symétrique de la définition.◊

La question qui vient directement à l'esprit est alors : existe-t-il un lien entre l'équivalence classique et l'équivalence intuitive ? On peut en fait prouver que ces deux notions se recouvrent complètement.

Proposition 4.10 (Lien entre équivalence intuitive et classique)

Quels que soient a et a' appartenant à $\mathbb{D}^{\#}$, on a toujours que a est classiquement équivalent à a' si et seulement si a est intuitivement équivalent à a', en bref

$$a \approx a' \Leftrightarrow a \equiv a'.$$

Preuve :

Il est évident que $a \equiv a'$ implique que $a \approx a'$, en effet la bijection (et sa réciproque) entre les locations intervenant dans l'équivalence intuitive convient naturellement comme fonction de mise en correspondance pour l'approximation.

L'implication inverse est moins évidente. Prenons donc comme hypothèses $a \leq^* a'$ et $a' \leq^* a$, et tentons de prouver $a \equiv a'$.

Remarquons d'abord que si A et B sont des ensembles finis et f et g des fonctions telles que

$$f : A \nrightarrow\!\!\!\!\!\rightarrow B \text{ et } g : B \nrightarrow\!\!\!\!\!\rightarrow A,$$

on a nécessairement que f et g sont des bijections.

En effet, puisqu'il existe une fonction surjective de A dans B, on a que $\#B \leq \#A$. De même, puisqu'il existe une fonction surjective de B dans A, on a que $\#A \leq \#B$. Et donc, $\#A = \#B$. Par conséquent, puisque f est une fonction surjective entre deux ensembles finis de même cardinal, f ne peut être qu'une bijection. Il en va évidemment de même pour g.

Montrons maintenant que

$$a \xrightarrow{f} a' \xrightarrow{g} a \Rightarrow g \circ f = id_{dom(a_1)}.$$

Il nous faut donc prouver, à partir de $a \xrightarrow{f} a' \xrightarrow{g} a$, que

$$\forall l \in dom(a_1),\ g(f(l)) = l.$$

Il est évident que la remarque précédente s'applique à f et g (puisque celles-ci sont surjectives par la proposition 4.6) et que ces deux fonctions ne peuvent être que des bijections.

On procède par récurrence sur les niveaux du domaine.

1. Le cas de base est trivial :

$$\forall i : 1 \leq i \leq n+1 : g(f(\alpha_i)) = g(\alpha'_i) = \alpha_i.$$

2. Supposons que

$$\forall l \in A_j,\ g(f(l)) = l,$$

et montrons que

$$\forall l \in A_j \cup \{Im(p_2(a_1(l_0))) \mid l_0 \in A_j \wedge a_1(l_0) \in \mathbb{I}nst^{\#}\},\ g(f(l)) = l.$$

Soit donc l tel que

$$\begin{cases} a_1(l_0) = (nc, va) \in \mathbb{I}nst^{\#}, \\ l_0 \in A_j, \\ va(ch) = l \wedge ch \in dom(va). \end{cases}$$

Montrons que $g(f(l)) = l$.

Comme g est une bijection, il existe une et une seule location l' telle que $g(l') = l_0$. D'autre part, puisque l_0 appartient à A_j, on a que $g(f(l_0)) = l_0$. Par conséquent, il faut que $f(l_0) = l'$.

Appliquons maintenant les hypothèses d'approximations.

Puisque $a' \xrightarrow{g} a$,

$$a_1(l_0) = a_1(g(l')) = (nc, va) \Rightarrow \begin{cases} a'_1(l') = (nc, va') \\ va(ch) = g(va'(ch)) \end{cases}$$

Puisque $a \xrightarrow{f} a'$,

$$a_1(l') = a_1(f(l_0)) = (nc, va') \Rightarrow \begin{cases} a_1(l_0) = (nc, va'') = (nc, va) \\ va'(ch) = f(va(ch)) \end{cases}$$

On peut donc écrire les égalités suivantes.

$$l = va(ch) = g(va'(ch)) = g(f(va(ch)) = g(f(l))$$

A partir de là, on peut facilement montrer que f convient comme bijection pour l'équivalence intuitive (on ne montre ici qu'un seul cas, les autres sont similaires).

$a'_1(f(l)) = t \in Type \Rightarrow a_1(l) = t$ puisque $a \xrightarrow{f} a'$

$a_1(l) = a_1(g(f(l)) = t \in Type \Rightarrow a'_1(f(l)) = t$ puisque $a' \xrightarrow{g} a$

$\Diamond$

4.8 Borne supérieure

4.8.1 Rappels

Pour débuter cette section, précisons ce que nous entendons exactement par *majorant* et *borne supérieure.*

Définition 4.11 (Majorant)

Soit E un espace muni d'un préordre $\leq$ et soit S une partie de E, on appelle majorant de S, tout élément e de E tel que

$\forall s \in S,\ s \leq e.$

Définition 4.12 (Borne supérieure)

Soit E un espace muni d'un préordre $\leq$ et soit S une partie de E, on appelle borne supérieure de S, tout élément e de E tel que

1. *e est un majorant de S,*
2. *$\forall e' \in E$, e' majorant de $S \Rightarrow e \leq e'$.*

Une borne supérieure de deux éléments du domaine doit donc, d'une part, approximer ces deux éléments et, d'autre part, garder "le plus d'information possible" sur ceux-ci. Au niveau des fonctions entre locations associées aux approximations, on dira que la fonction d'approximation relative à tout majorant doit pouvoir "se factoriser" au travers de la fonction relative à une borne supérieure.

4.8.2 Exemples

Maintenant que cette notion de borne supérieure est définie, différentes questions se posent. Est-ce que la borne supérieure existe toujours ? Peut-on trouver un algorithme qui la calcule ?

Pour pouvoir répondre à ces questions, nous choisissons de d'abord présenter sur quelques exemples ce qu'on aimerait intuitivement obtenir comme borne supérieure.

La figure 4.2 représente une situation (i.e. un élément de $\mathbb{D}^{\#}$), nommée $S1$, où interviennent troix variables de type statique A : x, y et z. Le type B est une spécialisation du type A. Une caractéristique de $S1$ est l'absence de "sharing". Toutes les situations concrètes correspondant à $S1$ ne présentent donc aucun point de partage. La figure 4.3 présente une situation $S2$ "comparable" à $S1$ (i.e. de même environnement) mais où existent des points de partage. Au niveau des types, $S1$ et $S2$ sont identiques.

Puisque $S1$ et $S2$ donnent les mêmes informations de type, il n'y a aucune raison que la borne

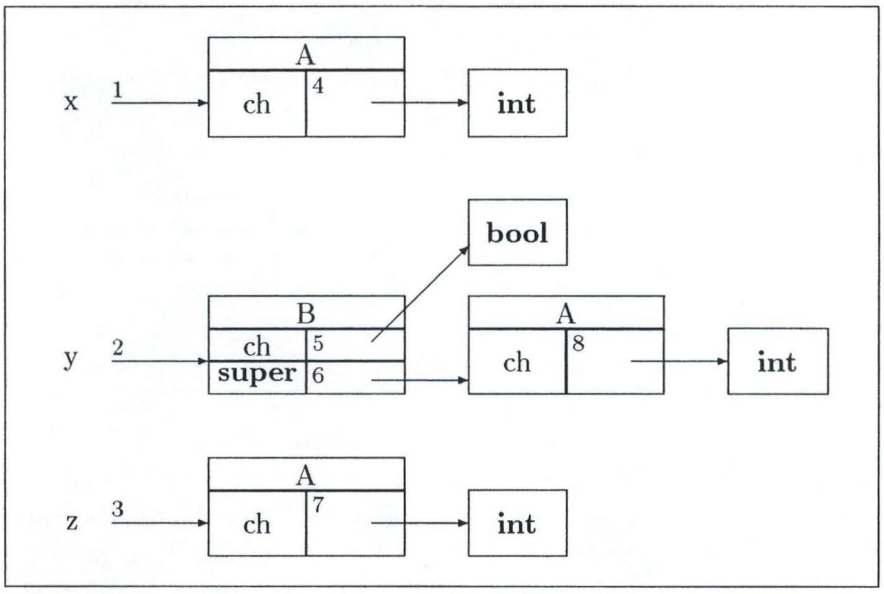

Figure 4.2: Situation $S1$

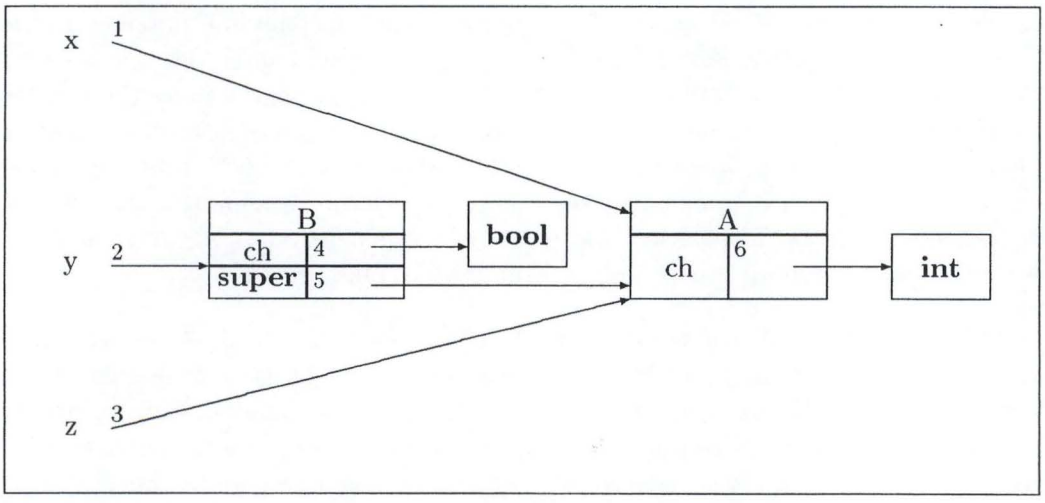

Figure 4.3: Situation $S2$

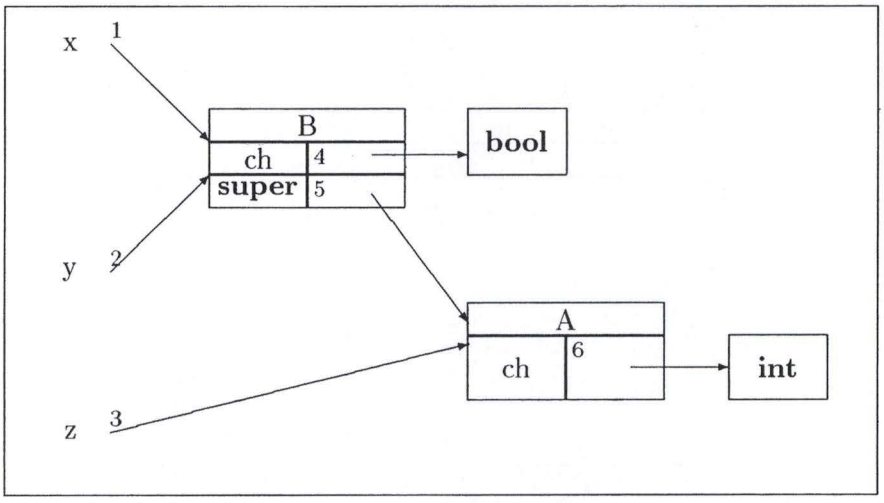

Figure 4.4: Situation $S3$

supérieure perde de l'information à ce sujet. De plus, $S1$ ne contient aucun point de partage pouvant entrer en conflit avec le "sharing" de $S2$. Il semble donc qu'une borne supérieure acceptable pour $S1$ et $S2$ soit tout simplement $S2$. D'un autre côté, si on regarde nos définitions, on constate que $S2$ approxime $S1$ et qu'il est donc logique que $S2$ soit la (une) borne supérieure.

Considérons maintenant une troisième situation $S3$ toujours relative au même environnement. Dans cette situation le type dynamique de x change. De plus, $S3$ présente des points de partage différents de ceux de $S2$.

Intéressons-nous maintenant à la borne supérieure de $S2$ et $S3$. Dans $S2$, le type de x est A et, dans $S3$, ce type est B. Par conséquent, la borne supérieure ne peut conserver qu'une approximation du type de x (on prendra l'approximation la plus fine possible à savoir $\alpha(A) \sqcup \alpha(B)$, soit $\{A, B\}$ en considérant $Type^{\#} = \wp(Type)$). Par contre, les types de y et z sont identiques dans les deux situations, pourquoi alors ne pas conserver toute l'information sur ces deux variables ? Dans $S3$, on ne sait pas si y.**super** et z désignent des instances distinctes, on ne doit donc pas non plus le savoir dans une approximation de $S2$ et $S3$. Toutes ces considérations nous amènent à proposer comme borne supérieure, la situation schématisée par la figure 4.5. On vérifie facilement que cette situation approxime effectivement $S1$ et $S2$ au sens de la définition 4.8 et on voit mal comment fournir une approximation plus précise.

Attaquons-nous à un dernier exemple. Les situations $S4$ et $S5$, symbolisées respectivement par les figures 4.6 et 4.7, manipulent deux variables de type B. Le type B possède un seul champ ch déclaré de type C. Ce dernier type peut être étendu par le type D. Dans la situation $S4$, les structures de x et y sont disjointes et le champ ch de x est de type C tandis que le champ ch de y est de type D. Dans la situation $S5$, x et y désignent la même instance abstraite dont le champ ch est de type D.

Dans $S5$, on ignore si x et y sont distinctes. Il est donc logique qu'il en soit de même pour une approximation de $S4$ et $S5$. Dans $S4$, $X.ch$ est de type C tandis que, dans $S5$, ce désignateur

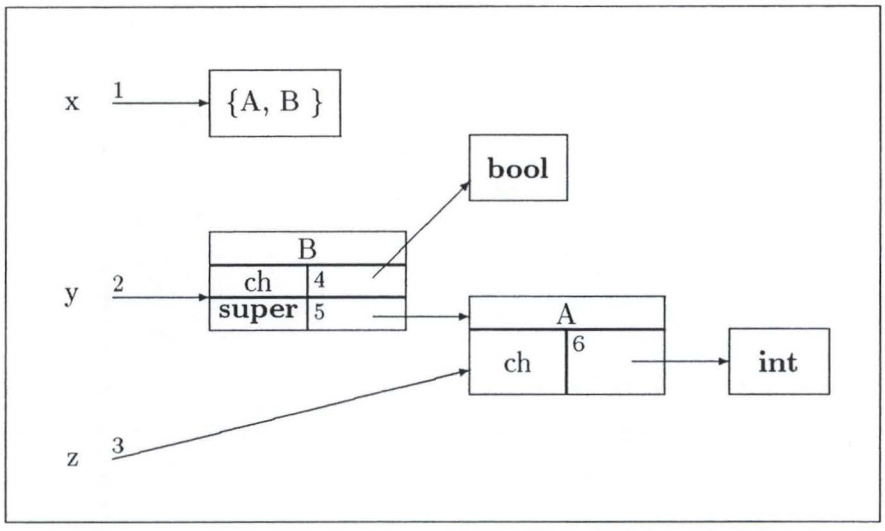

Figure 4.5: Borne supérieure de $S2$ et $S3$

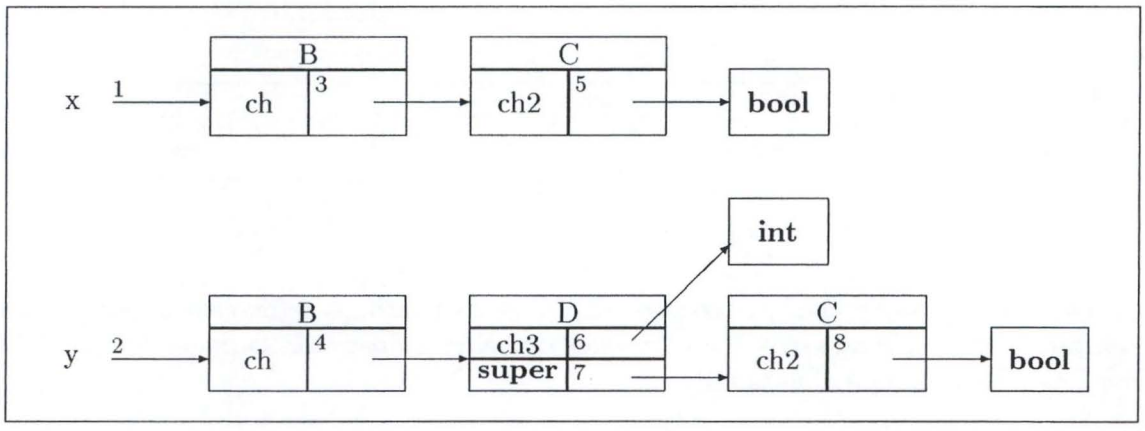

Figure 4.6: Situation $S4$

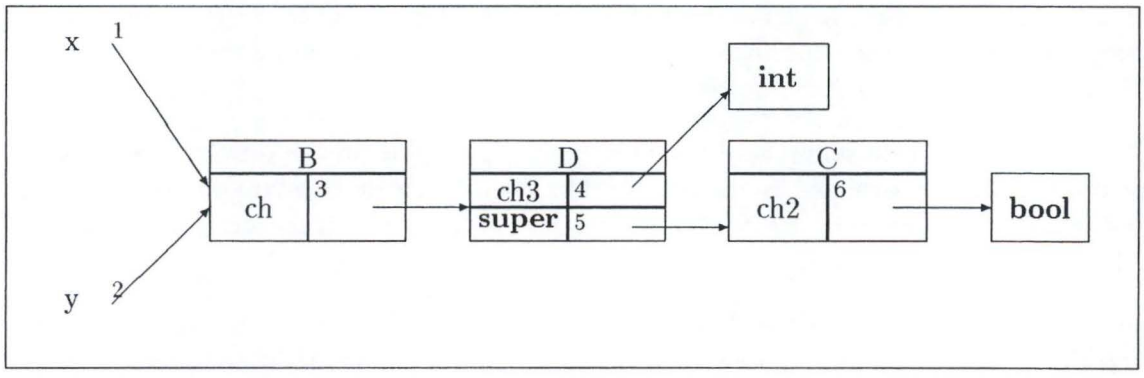

Figure 4.7: Situation $S5$

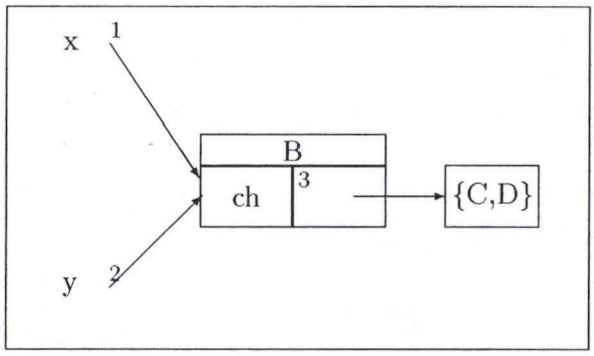

Figure 4.8: Une borne supérieure potentielle pour $S4$ et $S5$: $S6$

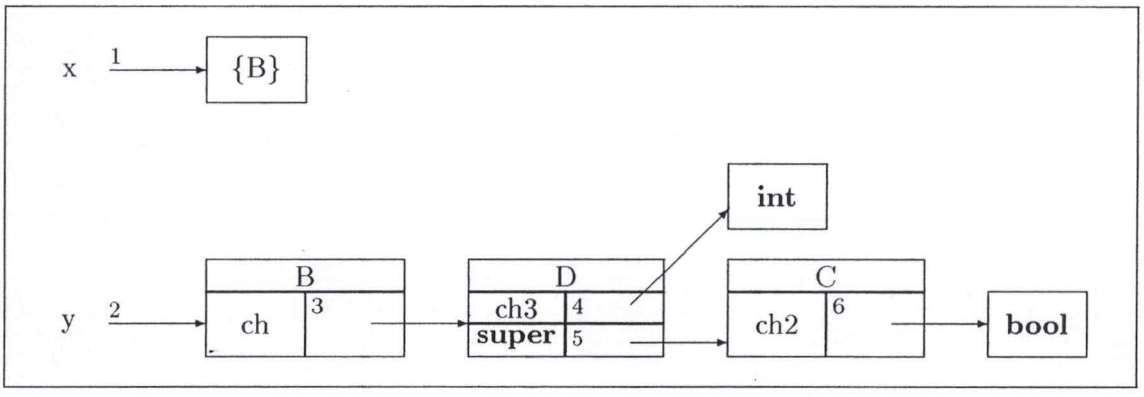

Figure 4.9: Une borne supérieure potentielle pour $S4$ et $S5$: $S7$

est de type D : on ne peut par conséquent qu'approximer le type de celui-ci par $\{C, D\}$. Ces constatations nous conduisent à la borne supérieure $S6$ reprise dans la figure 4.8 qui vérifie bien notre définition d'approximation.

On pourrait cependant être tenté de raisonner différemment. La structure de y est absolument identique dans $S4$ et $S5$. Par conséquent, il est logique que la borne supérieure conserve celle-ci. D'un autre côté, dans $S5$, x et y désignent la même instance et, dans $S4$, la structure de x n'est pas compatible avec la structure de y, abstrayons donc x. Ce raisonnement aboutit à proposer comme borne supérieure la situation $S7$ dessinée dans la figure 4.9. Tout comme $S6$, $S7$ approxime bien $S4$ et $S5$ au sens de la définition 4.8.

Même si, à vue d'oeil, on est tenté de croire que $S7$ est plus précise que $S6$, ces deux situations sont en fait incomparables. Nous sommes donc bien forcés de conclure que la borne supérieure n'existe pas toujours dans $(\mathbb{D}^{\#}, \leq^*)$. On obtient en fait "des majorants minimaux" à la place "d'un majorant minimum".

Intuitivement, cette existence de plusieurs "bornes incomparables" provient du fait que nous traitons deux types d'approximations simultanément : l'approximation située directement au niveau des types et l'approximation intervenant au niveau du "sharing".

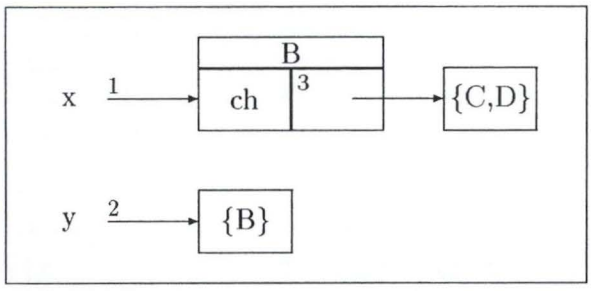

Figure 4.10: Situation $S8$

Remarquons que lorsque nous abstrayons un type, comme c'est le cas pour x dans $S7$, nous ne nous soucions absolument plus des informations relatives au "sharing" : $S7$ donne des valeurs différentes pour x et y alors que $S5$ exprime qu'il se peut que x et y désignent la même instance. Une contrainte intuivement acceptable serait justement de rejeter ce type d'approximation ($S7$ ne serait alors plus un majorant pour $S4$ et $S5$, la borne supérieure étant alors $S6$). La dernière section de ce chapitre présente les bases d'un domaine incorporant cette contrainte.

Si on peut choisir de privilégier l'optique sous-jacente à l'approximation réalisée par $S6$, pourquoi, d'un autre côté, ne pas privilégier l'optique sous-jacente à $S7$? Le problème est que lorsque nous avons construit $S7$, nous avons arbitrairement décidé de conserver la structure de y (qui *de visu* est ici plus imposante que celle de x). Cependant, en toute généralité, on pourrait aussi bien décider de conserver "le plus possible" la structure de x. Cette décision nous mènerait à la situation $S8$ reprise dans la figure 4.10 (cette situation est moins précise que $S6$ mais incomparable avec $S7$).

Cette dernière remarque, nous indique que, bien qu'il soit sans doute facile de définir un domaine rejetant le majorant $S6$, il serait sans doute relativement complexe d'obtenir, dans un tel domaine, l'existence de la borne supérieure. C'est pourquoi nous choisissons, pour l'instant, de ne pas aller plus loin dans cette voie.

4.9 Modification du domaine

Dans cette section, nous ébauchons la définition d'un raffinement de $\mathbb{D}^{\#}$ qui tient compte de la remarque évoquée lors de la "construction" des "bornes supérieures" de $S4$ et $S5$ au sujet de la conservation du "sharing". Il s'agit d'inclure dans le domaine une contrainte exprimant que si on observe, à un moment donné, un point de partage, quelles que soient les abstractions effectuées par la suite, on conserve ce point de partage. Pour ce faire, l'idée est d'enrichir le domaine de sorte à permettre la modélisation du "sharing" aussi au niveau des élements "abstraits".

Nous espérons qu'un tel domaine permettra la définition et la construction d'une borne supérieure "unique".

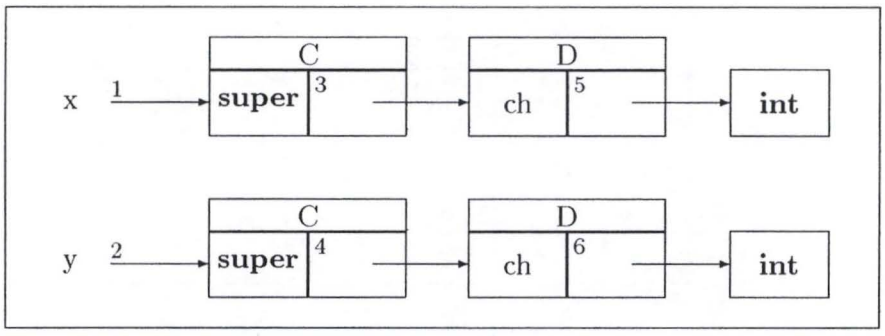

Figure 4.11: Situation $S1$

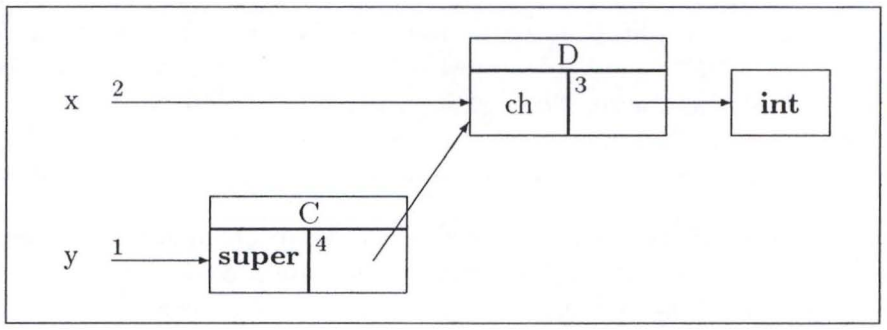

Figure 4.12: Situation $S2$

4.9.1 Elargissement de $\mathbb{D}^\#$

Avant d'aborder la définition de ce "nouveau domaine", remarquons que l'idée même de "conservation du sharing" nous pousse à élargir[2] la définition de $\mathbb{D}^\#$.

En effet, en anticipant quelque peu sur la suite, considérons les situations abstraites suivantes : $S1$ et $S2$, respectivement symbolisées par les figures 4.11 et 4.12, manipulent deux variables x et y déclarées de type D, qui admet une extension C.

[2]Le terme élargissement n'est évidemment pas ici à prendre au sens technique habituel d'opérateur de "widening".

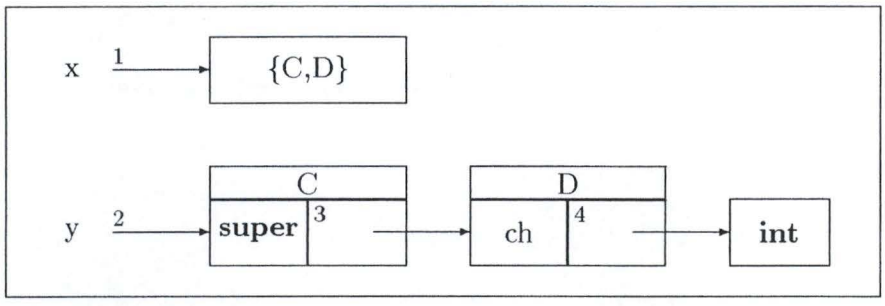

Figure 4.13: Situation $S3$

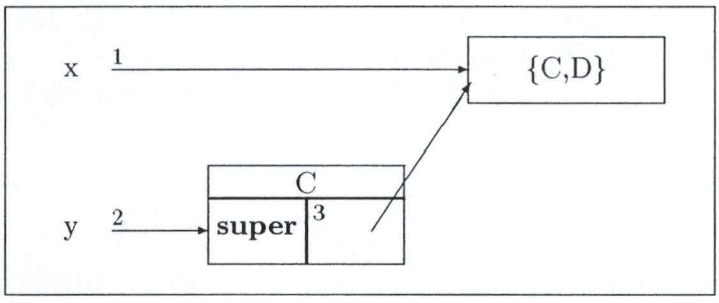

Figure 4.14: Situation $S4$

Dans $S1$, x et y désignent deux instances distinctes de type C, tandis que, dans $S2$, y désigne une instance de type C et x désigne l'instance "père" de celle-ci. Dans $(\mathbb{D}^{\#}, \leq^{*})$, nous aurions intuitivement proposé comme borne supérieure la situation $S3$ représentée par la figure 4.13.

Or la situation $S3$ est justement une situation que nous désirons éliminer de l'ensemble des approximations "valides" pour $S2$: en effet, dans $S2$, x et y.**this** désignent la même instance abstraite et nous souhaitons donc, dans $\mathbb{D}^{\#1}$, que toute approximation de $S2$ associe la même valeur à x et y.**super**.

L'obligation d'associer la même valeur à x et y.**super** nous conduit, soit à perdre toute la structure de y (ce qui serait ici un peu stupide), soit à permettre d'approximer la valeur de y.**super** par $\{C, D\}$. Cette dernière optique est représentée par la figure 4.14. Remarquons que $S4$ n'appartient absolument pas à $\mathbb{D}^{\#}$ (tel que nous l'avons défini jusqu'à présent).

Ce petit exemple nous montre que notre vision "binaire" de la conservation des instances impliquant une conservation totale de la structure relative à l'héritage est sans doute trop exigeante. C'est pourquoi nous élargissons la définition de $\mathbb{D}^{\#}$ comme suit.

Propriété 1 : Forme de l'environnement
Propriété 2 : Unicité des locations
Propriété 3 : Coupure du store

Pour ces trois propriétés, on se référera à la page 86.

Propriété 4 : Respect des types statiques

Si $a_1(l) = (nc, v)$ est un élément de $\mathbb{I}nst^\#$, le store doit vérifier les conditions ci-après.

- $\mathbf{super} \in dom(v) \Leftrightarrow nc \in dom(\pi)$
- $\forall ch \in Nchamp, ch \in dom(v) \Leftrightarrow ch \in dom(\epsilon^0_{ch} nc)$
- $\mathbf{super} \in dom(v) \Rightarrow \begin{cases} \mathbf{soit} & a_1(v(\mathbf{super})) = e \in Type^\# \ \wedge \ \pi(nc) \in \gamma(e) \\ \mathbf{soit} & a_1(v(\mathbf{super})) \in \mathbb{I}nst^\# \ \wedge \ p_1(a_1(v(\mathbf{super}))) = \pi(nc) \end{cases}$
- $ch \in dom(v) \Rightarrow \begin{cases} \mathbf{soit} & a_1(v(ch)) \in Type \ \wedge \ a_1(v(c)) \preceq_\pi \epsilon^0_{ch} \ nc \ ch \\ \mathbf{soit} & a_1(v(ch)) = e \in Type^\# \ \wedge \ \exists t \in \gamma(e), t \preceq_\pi \epsilon^0_{ch} \ nc \ ch \\ \mathbf{soit} & a_1(v(ch)) \in \mathbb{I}nst^\# \ \wedge \ p_1(a_1(v(ch))) \preceq_\pi \epsilon^0_{ch} \ nc \ ch \end{cases}$

On a également.

$$1 \leq i \leq n \Rightarrow \begin{cases} \mathbf{soit} & a_1(\alpha_i)) = t \in Type \ \wedge \ t \preceq_\pi t_i \\ \mathbf{soit} & a_1(\alpha_i) = e \in Type^\# \ \wedge \ \exists t \in \gamma(e), t \preceq_\pi t_i \\ \mathbf{soit} & a_1(\alpha_i) \in \mathbb{I}nst^\# \ \wedge \ p_1(a_1(\alpha_i)) \preceq_\pi t_i \end{cases}$$

$$\begin{cases} \mathbf{soit} & a_1(\alpha_{n+1}) = t \in Type \ \wedge \ t = t_{n+1} \\ \mathbf{soit} & a_1(\alpha_{n+1}) = e \in Type^\# \ \wedge \ t_{n+1} \in \gamma(e) \\ \mathbf{soit} & a_1(\alpha_{n+1}) \in \mathbb{I}nst^\# \ \wedge \ p_1(a_1(\alpha_{n+1})) = t_{n+1} \end{cases}$$

Définition 4.13 (Domaine d'interprétation $\mathbb{D}^\#$)

$\mathbb{D}^\#$ est défini par la relation d'appartenance suivante : a appartient à $\mathbb{D}^\#$ si et seulement si a est un élément du produit $\mathbb{E}nv^\# \times \mathbb{S}tore^\#$ qui vérifie les propriétés 1, 2, 3 et 4.

4.9.2 Ebauche d'un nouveau domaine

Passons maintenant au "nouveau domaine" à proprement parler. Rappelons que le but de ce domaine est de permettre de "raffiner" le préordre défini sur $\mathbb{D}^\#$ (i.e. on veut que certains éléments comparables dans $(\mathbb{D}^\#, \leq^*)$ ne le soient plus).

Nous avons besoin de pouvoir modéliser une notion de partage non seulement au niveau des instances abstraites mais également au niveau des valeurs de $Type^\#$. Jusqu'à présent, nous avons toujours traduit le partage au niveau des locations : deux instances sont identiques si les locations attribuées à leurs champs sont identiques. Pourquoi ne pas procéder de la même manière pour les éléments "abstraits" ? Nous adjoignons donc une location abstraite aux éléments de $Type^\#$ apparaissant dans le store. Ces locations sont toutes associées par le store à une valeur factice **undef**.

Nous parlons "d'ébauche de domaine" car certaines définitions et certaines propriétés ne sont pas encore réellement approfondies et surtout parce que nous ne sommes pas encore certains que ce domaine permette la définition d'un opérateur de borne supérieure.

4.9.2.1 Définitions

On obtient les nouvelles définitions suivantes.

Définition 4.14 (Valeurs abstraites)

Le nouvel ensemble des valeurs abstraites est noté $\mathbb{V}al^{\#1}$. *Il est défini comme l'union disjointe*

$$\mathbb{V}al^{\#1} = Type + Type^{\#} \times \mathbb{L}oc^{\#} + \mathbb{I}nst^{\#}.$$

Définition 4.15 (Store)

$\mathbb{S}tore^{\#1}$ *désigne l'ensemble des stores abstraits.*

$$\mathbb{S}tore^{\#1} = \mathbb{L}oc^{\#} \rightarrowtail \mathbb{V}al^{\#1} + \{\mathbf{undef}\}$$

Passons maintenant aux propriétés caractéristiques du domaine.

Propriété 1 : Forme de l'environnement (inchangée)

Propriété 2 : Unicité des locations

$a_1(l) = (nc, v) \in \mathbb{I}nst^{\#} \wedge a_1(l') = (nc', v') \in insta \Rightarrow (a_1(l) = a_1(l') \vee Im(v) \cap Im(v') = \emptyset)$
$a_1(l) = (e, l) \in Type^{\#} \times \mathbb{L}oc^{\#} \wedge a_1 = (nc, v) \in \mathbb{I}nst^{\#} \Rightarrow l \notin Im(v)$

Propriété 3 : Coupure du store

Si on définit la suite de niveaux (A_j) de limite A par

$$A_0 = \{\alpha_i \mid 1 \leq i \leq n+1\}$$

$$A_{j+1} = A_j \cup \{p_2(a_1(l)) \mid l \in A_j \wedge a_1(l) \in Type^{\#} \times \mathbb{L}oc^{\#}\} \cup \bigcup_{\substack{l \in A_j \\ a_1(l) \in \mathbb{I}nst^{\#}}} Im(p_2(a_1(l)))$$

on doit avoir l'égalité

$$A = dom(a_1).$$

Propriété 4 : Respect des types statiques

(Inchangée si ce n'est qu'il faut naturellement remplacer les éléments de $Type^{\#}$ par des éléments de $Type^{\#} \times \mathbb{L}oc^{\#}$)

Propriété 5 : Propriété caractéristique

$$a_1(l) = (e, loc) \in Type^{\#} \times \mathbb{L}oc^{\#} \Rightarrow a_1(loc) = \textbf{undef}$$

$$a_1(l) = (nc, v) \in \mathbb{I}nst^{\#} \Rightarrow \forall loc \in Im(v),\ a_1(loc) \neq \textbf{undef}$$

Définition 4.16 (Domaine d'interprétation abstrait)

Le nouveau domaine d'interprétation abstrait est noté $\mathbb{D}^{\#1}$ *et est défini par la relation d'appartenance suivante : a appartient à* $\mathbb{D}^{\#1}$ *si et seulement si a est un élément du produit* $\mathbb{E}nv^{\#} \times \$tore^{\#1}$ *qui vérifie les propriétés 1, 2, 3, 4 et 5.*

Passons maintenant à la définition de la fonction de concrétisation. Celle-ci est évidemment presque identique à la définition précédente. Il faut juste ajouter une contrainte au niveau des abstractions d'instances : la valeur abstraite, maintenant "identifiable", doit être la seule valeur correspondant à l'instance dans le store abstrait.

Définition 4.17 ($d \xrightarrow{(f,+)} a$)

Soient a appartenant à $\mathbb{E}nv^{\#} \times \$tore^{\#1}$ *et d appartenant à* $\mathbb{E}nv \times \$tore$.
On dira que a approxime strictement d au travers de f, i.e. $d \xrightarrow{(f,+)} a$, *si et seulement si a, d et f vérifient les trois propriétés ci-dessous.*

1. $f : dom(d_1) \rightarrowtail dom(a_1)$
2. $d_0 = \perp_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \textbf{this}/t_{n+1}\delta_{n+1}]$
   $a_0 = \perp_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \textbf{this}/t_{n+1}\alpha_{n+1}]$
   $\forall i : 1 \leq i \leq n+1 : f(\delta_i) = \alpha_i$
3. $\forall l \in dom(f)$,

$$a_1(f(l)) = t \in Type \Rightarrow d_1(l) \in \mathbb{B}ase \wedge p_1(d_1(l)) = t$$

$$a_1(f(l)) = (e, loc) \in Type^{\#} \times \mathbb{L}oc^{\#} \Rightarrow \begin{cases} \textbf{soit } d_1(l) = (nc, v) \in \mathbb{I}nst \wedge nc \in \gamma(e) \wedge \forall l \in Im(v),\ f(l) = loc \\ \textbf{soit } d_1(l) \in \mathbb{B}ase \wedge p_1(d_1(l)) \in \gamma(e) \end{cases}$$

$$a_1(f(l)) = (nc, va) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} d_1(l) = (nc, v) \in \mathbb{I}nst \\ dom(v) = dom(va) \\ \forall ch \in dom(v),\ va(ch) = f(v(ch)) \end{cases}$$

Remarquons que l'introduction quelque peu artificielle des locations au niveau des valeurs abstraites primitives induit, du moins dans notre formulation, la perte de la surjectivité de la fonction de mise en correspondance f : en effet, lorsqu'on abstrait un élément de $\mathbb{B}ase$, la location abstraite intervenant dans la valeur abstraite n'a pas nécessairement de contrepartie concrète.

Définition 4.18 (Fonction de concrétisation)

La fonction de concrétisation se note γ^{*1} *et est définie comme suit.*

$$\begin{array}{rccl} \gamma^{*1} : & \mathbb{D}^{\#1} & \longrightarrow & \wp(\mathbb{D}^*) \\ & a & \rightsquigarrow & \{d \mid \exists f,\ d \xrightarrow{(f,+)} a\} \end{array}$$

On introduit au niveau de la définition du préodre des modifications tout à fait similaires à celles introduites pour la fonction de concrétisation.

Définition 4.19 ($a \xrightarrow{(f,+)} a'$)

Soient a *et* a' *appartenant à* $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
On dira que a' *approxime strictement* a *au travers de* f, *i.e.* $a \xrightarrow{(f,+)} a'$, *si et seulement si* a, a' *et* f *vérifient les trois propriétés ci-dessous.*

1. $f : dom(a_1) \rightarrowtail dom(a'_1)$

2. $a_0 = \bot_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $a'_0 = \bot_0^{\#}[v_1/t_1\alpha'_1, ..., v_n/t_n\alpha'_n, \mathbf{this}/t_{n+1}\alpha'_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : f(\alpha_i) = \alpha'_i$

3. $\forall l \in dom(f)$,

$$a'_1(f(l)) = t \in Type \Rightarrow a_1(l) = t$$

$$a'_1(f(l)) = (e', loc') \in Type^{\#} \times \mathbb{L}oc^{\#} \Rightarrow \left\{ \begin{array}{l} \textbf{soit } a_1(l) \in Type \wedge a_1(l) \in \gamma(e') \\ \textbf{soit } a_1(l) = (e, loc) \in Type^{\#} \times \mathbb{L}oc^{\#} \wedge e \leq e' \wedge f(loc) = loc' \\ \textbf{soit } a_1(l) = (nc, v) \in \mathbb{I}nst^{\#} \wedge nc \in \gamma(e') \wedge \forall ll \in Im(v),\ f(ll) = loc' \end{array} \right.$$

$$a'_1(f(l)) = (nc, v') \in \mathbb{I}nst^{\#} \Rightarrow \left\{ \begin{array}{l} a_1(l) = (nc, v) \in \mathbb{I}nst^{\#} \\ dom(v) = dom(v') \\ \forall ch \in dom(v),\ v'(ch) = f(v(ch)) \end{array} \right.$$

Définition 4.20 (Préordre sur $\mathbb{D}^{\#}$)

Soient deux éléments a *et* a' *de* $\mathbb{D}^{\#1}$. *On dira que* a' *approxime strictement* a *(ce qu'on note* $a \leq^{*1} a'$*) si et seulement si il existe une fonction* f *telle que* $a \xrightarrow{(f,+)} a'$.

On a toujours les deux théorèmes suivants. Les preuves de ceux-ci reposent principalement sur des mécanismes de compositions de fonctions.

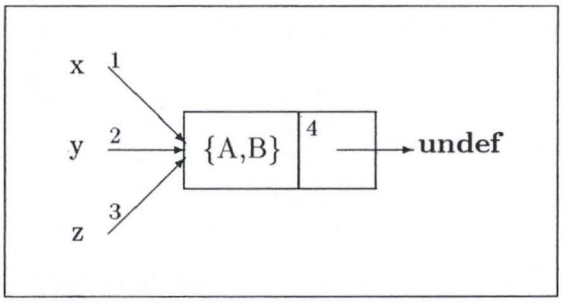

Figure 4.15: Borne supérieure de $S2$ et $S3$

Proposition 4.11 (Relation de préordre)

La relation $\leq^*$ définie sur $\mathbb{D}^\#$ est effectivement une relation de préordre, i.e. elle est réflexive et transitive.

Proposition 4.12 (Cohérence de $\leq^{*1}$ par rapport à γ^{*1})

Soient a et a' appartenant à $\mathbb{D}^{\#1}$. Si a' approxime a, la concrétisation de a' contient la concrétisation de a, i.e.

$$a \leq^{*1} a' \Rightarrow \gamma^{*1}(a) \subseteq \gamma^{*1}(a')$$

4.9.2.2 Exemples

Reconsidérons les exemples présentés au cours de la section précédente (cf. page 102) et voyons si nous obtenons bien les résultats escomptés.

La borne supérieure relative aux situations $S1$ et $S2$ ne fait intervenir aucune approximation de types et reste par conséquent inchangée.

En ce qui concerne la borne supérieure de $S2$ et $S3$, nous devons par contre revoir notre construction : dans $S2$, x est de type A tandis que, dans $S3$, x est de type B. On doit donc approximer le type de x par $\{A, B\}$. D'autre part, dans $S2$, x et z désignent la même instance. Par conséquent, z doit subir la même approximation que x. Il en va de même pour y puisque x et y désignent la même instance dans $S3$. On obtient par conséquent la borne supérieure représentée par la figure 4.15. Si on "oublie" la location identifiante de la valeur abstraite, il est clair que cette situation est un majorant de $S2$ et $S3$ au sens de $\mathbb{D}^\#$ et n'est certainement pas "l'approximation la plus fine possible". La contrainte que nous ajoutons au niveau de $\mathbb{D}^{\#1}$ peut donc être, assez logiquement, source de "perte d'information".

La borne supérieure de $S4$ et $S5$ est bien la situation $S6$ (pour peu évidemment qu'on adjoigne à la valeur abstraite $\{C, D\}$ une location identifiante).

4.9.2.3 Principes de l'algorithme

Nous ne donnons ici que les idées principales relatives à un éventuel algorithme de borne supérieure. Nous n'avons malheureusement pas eu le temps de rédiger complètement celui-ci et encore moins de formuler précisément les propriétés sous-jacentes à sa construction. Sans ces propriétés, il nous est impossible de prouver sa correction (remarquons qu'une telle preuve de correction prouverait évidemment l'existence d'une borne supérieure "unique").

Avant toute chose, signalons que nous avons la conviction que l'optique de "conservation du sharing" additionnée à l'existence possible de "cycles" au niveau des structures des champs empêche la construction d'un algorithme "simple" procédant "en une seule passe".

Nous proposons donc l'esquisse d'un algorithme en "deux passes" calculant la borne supérieure de deux éléments a et a' de $\mathbb{D}^{\#1}$ relatifs au même environnement. Cet algorithme "expérimental" est donné à titre d'indication : il est incomplet et probablement "erroné" dans le traitement de certains détails.

Première passe

On construit un couple "environnement/store" (b_0, b_1) dont les locations sont des couples de locations (la première composante appartient au domaine de a_1 et la seconde au domaine de a_2) sauf les "locations" associées aux éléments de $Type^{\#}$ qui sont des couples d'ensembles de locations.

$$b_0 : Nvar + \{\mathbf{this}\} \rightarrowtail Type \times (\mathbb{L}oc^{\#} \times \mathbb{L}oc^{\#})$$

$$b_1 : \mathbb{L}oc^{\#} \times \mathbb{L}oc^{\#} \rightarrowtail Type + Type^{\#} \times (\wp(\mathbb{L}oc^{\#}) \times \wp(\mathbb{L}oc^{\#})) + Nclasse \times (Nchamps + \{\mathbf{super}\} \rightarrowtail \mathbb{L}oc^{\#} \times \mathbb{L}oc^{\#})$$

Cette première passe se charge juste de "répercuter" sur le reste du store les associations de locations induites par l'environnement. Elle effectue également des approximations là où les structures sont "directement" incompatibles.

Le pseudo-algorithme correspondant à cette première passe est fourni dans la figure 4.16.

La fonction $\tilde{\alpha}$, employée dans cet algorithme, étend juste la portée de la fonction primitive d'abstraction α.

$$\begin{array}{lccll} \tilde{\alpha} : & |Val^{\#1} & \longrightarrow & Type^{\#} & \\ & (e, loc) & \rightsquigarrow & e & \text{si } (e, loc) \in Type^{\#} \times \mathbb{L}oc^{\#} \\ & t & \rightsquigarrow & \alpha(t) & \text{si } t \in Type \\ & (nc, v) & \rightsquigarrow & \alpha(nc) & \text{si } (nc, v) \in \mathbb{I}nst^{\#} \end{array}$$

begin
$b_0 := \bot_0[v_1/t_1(\alpha_1, \alpha'_1), ..., v_n/t_n(\alpha_n, \alpha'_n), v_{n+1}/t_{n+1}(\alpha_{n+1}, \alpha'_{n+1})]$
$b_1 := \bot_1$
$W := \{(\alpha_1, \alpha'_1), ..., (\alpha_{n+1}, \alpha'_{n+1})\}$
while $W \neq \emptyset$ **do**
  begin
    $W \rightarrow W + \{(l, l')\}$
    if $a_1(l) = a'_1(l') = t \in Type$
    then $b_1((l, l')) := t$
    else
      if $a_1(l) = (nc, va) \in \mathbb{I}nst^{\#} \wedge a'_1(l') = (nc, va') \in \mathbb{I}nst^{\#}$
        $\wedge\, dom(va) = \{x_1, ..., x_m\}$
      then
        begin
          $b_1((l, l')) := (nc, v)$
          for-each $i \in \{1, ..., m\}$ **do** $v(x_i) := (va(x_i), va'(x_i))$
          $W := W + \{(va(x_1), va'(x_i)), ..., (va(x_m), va'(x_m))\}$
        end
      else $b_1((l, l')) := (e, loc)$
        where $e = \tilde{\alpha}(a_1(l)) \sqcup \tilde{\alpha}(a'_1(l'))$
          $\wedge\, loc = (\mathcal{E}ns(a_1(l)), \mathcal{E}ns(a'_1(l')))$
  end
end

Figure 4.16: Première passe

La fonction $\mathcal{E}ns$ renvoie l'ensemble des locations "identifiant" une valeur.

$$\begin{array}{lccll} \mathcal{E}ns: & \mathbb{V}al^{\#1} & \longrightarrow & \wp(\mathbb{L}oc^{\#}) & \\ & (e, loc) & \rightsquigarrow & \{loc\} & \text{si } (e, loc) \in Type^{\#} \times \mathbb{L}oc^{\#} \\ & t & \rightsquigarrow & \emptyset & \text{si } t \in Type \\ & (nc, v) & \rightsquigarrow & Im(v) & \text{si } (nc, v) \in \mathbb{I}nst^{\#} \end{array}$$

La notation $W \to W + \{(l, l')\}$ correspond au retrait d'un élément arbitraire (l, l') de l'ensemble W.

Seconde passe

Si, dans le couple (b_0, b_1), il existe des instances de première projection identique, cela signifie que ces instances représentent "le même objet" pour a. De même, s'il existe des instances de seconde projection identique, elles représentent le même objet de a'. On peut étendre ces considérations aux valeurs abstraites.

La seconde passe a pour objectif de "fusionner" toutes les valeurs correspondant aux mêmes objets dans l'un ou l'autre des éléments dont on calcule la borne supérieure. Il est évident que ces fusions peuvent s'accompagner de "nouvelles approximations".

Nous ne détaillons pas l'algorithme relatif à cette seconde passe. Mentionnons seulement qu'il peut par exemple construire, à partir du couple (b_0, b_1), un couple (c_0, c_1) tel que

$$c_0 : Nvar + \{\mathbf{this}\} \rightarrowtail Type \times (\wp(\mathbb{L}oc^{\#}) \times \wp(\mathbb{L}oc^{\#}))$$

$$\begin{aligned} c_1 : \wp(\mathbb{L}oc^{\#}) \times \wp(\mathbb{L}oc^{\#}) \rightarrowtail\ & Type + Type^{\#} \times (\wp(\mathbb{L}oc^{\#}) \times \wp(\mathbb{L}oc^{\#})) \\ & + Nclasse \times (Nchamps + \{\mathbf{super}\} \rightarrowtail \wp(\mathbb{L}oc^{\#}) \times \wp(\mathbb{L}oc^{\#})) \end{aligned}$$

qui conserve au maximum l'information reprise dans (b_0, b_1) tout en "fusionnant" toutes les valeurs qui doivent l'être.

Le store c_1 devra par exemple vérifier la propriété :

$$\forall e, e' \in dom(c_1) : e \neq e' : p_1(e) \cap p_1(e') = \emptyset \wedge p_2(e) \cap p_2(e') = \emptyset.$$

Les ensembles de locations intervenant dans c_1 correspondent en fait aux fonctions de mise en correspondance entre a (respectivement a') et la "borne supérieure".

Chapitre 5

Domaine abstrait (version 2)

Au cours du chapitre 4, nous avons proposé un domaine abstrait dont les caractères fondamentaux étaient un traitement "binaire" de l'information au niveau des types et une interprétation du "sharing" en termes d'approximation.

Nous proposons dans ce chapitre un second domaine. Dans ce domaine, les approximations relatives aux types eux-mêmes sont toujours traitées de manière "binaire". La différence par rapport au domaine précédent ne peut par conséquent se situer qu'au niveau de l'interprétation du "sharing".

En effet, pourquoi ne pourrait-on pas envisager une interprétation duale pour le "sharing" et voir celui-ci non plus comme une perte d'information mais au contraire comme une conservation d'information ? Dans cette nouvelle optique, si le store abstrait associe à x et à y la même instance, au niveau concret x et y désignent effectivement la même instance.

La structure de ce chapitre est quasiment identique à celle des deux chapitres précédents si ce n'est qu'il n'est pas nécessaire de définir la forme du domaine puisqu'il s'agit simplement de $\mathbb{D}^{\#}$.

Nous définissons donc successivement les notions de fonction de concrétisation, de fonction d'abstraction et de préordre. Nous fournissons ensuite un algorithme de calcul pour la borne supérieure relative à ce préordre et nous terminons le chapitre par une petite illustration de "l'utilisation" des deux domaines.

Il peut paraître fastidieux et inutile de reconsidérer une à une ces différentes définitions et propriétes. Cependant, d'une part, il nous semble qu'un exposé mentionnant uniquement les changements effectués relativement aux premiers domaines manquerait de clarté et, d'autre part, nous risquerions ainsi que, justement, certaines différences "fondamentales" nous échappent.

5.1 Fonction de concrétisation

Comme nous l'avons signalé dans l'introduction, le domaine en lui-même est absolument identique au domaine défini au chapitre 4. Ce qui change est l'interprétation de ce domaine, i.e. sa fonction de concrétisation (notamment).

Nous employons le même schéma de définition que pour les domaines précédents et introduisons, avant de présenter la fonction de concrétisation elle-même, le concept d'approximation au travers d'une relation entre les locations concrètes et les locations abstraites.

Une caractéristique fondamentale de la relation de mise en correspondance précédente était son aspect fonctionnel : deux locations abstraites différentes ne correspondent jamais à la même location concrète. Au contraire, vu l'interprétation adoptée pour le "sharing", la relation de mise en correspondance de ce domaine doit être telle que deux locations concrètes différentes ne correspondent jamais à la même location abstraite, c'est-à-dire que la relation doit être injective.

Par facilité, nous choisissons de considérer les relations réciproques qui doivent donc être fonctionnelles.

Définition 5.1 ($d \overset{f}{\leftarrow} a$)

Soient a appartenant à $\mathbb{E}nv^{\#} \times \$tore^{\#}$ et d appartenant à $\mathbb{E}nv \times \$tore$.
On dira que a approxime d au travers de f, i.e. $d \overset{f}{\leftarrow} a$, si et seulement si a, d et f vérifient les trois propriétés ci-dessous.

1. $f : dom(a_1) \rightarrowtail dom(d_1)$
2. $d_0 = \perp_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \mathbf{this}/t_{n+1}\delta_{n+1}]$
   $a_0 = \perp_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $\forall i : 1 \leq i \leq n+1 : f(\alpha_i) = \delta_i$
3. $\forall l \in dom(f)$,

$$a_1(l) = t \in Type \Rightarrow d_1(l) \in \mathbb{B}ase \wedge (p_1(d_1(f(l))) = t$$
$$a_1(l) = e \in Type^{\#} \Rightarrow p_1(d_1(f(l))) \in \gamma(e)$$
$$a_1(l) = (nc, va) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} d_1(f(l)) = (nc, v) \in \mathbb{I}nst \\ dom(v) = dom(va) \\ \forall ch \in dom(v),\ f(va(ch)) = v(ch) \end{cases}$$

Il était logique, dans l'ancien domaine, que les relations de mise en correspondance soient partielles puisqu'on peut, en abstrayant des types, perdre certaines structures. Pour la même raison, il est tout aussi logique que les relations de mise en correspondance de ce domaine ne soient pas surjectives.

Il était également naturel, dans l'ancien domaine, que la relation soit surjective (des locations

abstraites sans contreparties concrètes n'ont pas de sens). De même, il semble naturel d'avoir ici une mise en correspondance partout définie. C'est ce qu'exprime la proposition 5.1, qui n'est autre que la propositon "duale" de la proposition 4.2.

Proposition 5.1 (Domaine de la fonction de mise en correspondance)

Soient d appartenant à $\mathbb{E}nv \times \$tore$ et a appartenant à $\mathbb{E}nv^{\#} \times \$tore^{\#}$.
Si $d \overset{f}{\leftarrow} a$, on a l'implication : $a \in \mathbb{D}^{\#} \Rightarrow f$ partout définie.

Preuve :

On montre facilement que $d \overset{f}{\leftarrow} a$ implique

$$\forall i \geq 0,\ A_i \subseteq dom(f).$$

En effet, il est évident que $A_0 = \{\alpha_1, ..., \alpha_{n+1}\} \subseteq dom(f)$ (propriété 2 de la définition 5.1). De plus si $a_1(l) = (nc, va) \in \mathbb{I}nst^{\#}$ et $ll = va(ch)$, la propriété 3 de la définition 5.1 implique directement que ll appartient au domaine de f.

A partir de là, on peut écrire

$$dom(a_1) = A = A_n \subseteq dom(f),$$

ce qui exprime bien que f est partout définie.$\Diamond$

Définition 5.2 (Fonction de concrétisation)

La fonction de concrétisation se note γ^{*2} et est définie comme suit.

$$\begin{array}{rccl} \gamma^{*2}: & \mathbb{D}^{\#} & \longrightarrow & \wp(\mathbb{D}^{*}) \\ & a & \rightsquigarrow & \{d \mid \exists f,\ d \overset{f}{\leftarrow} a\} \end{array}$$

5.2 Fonction d'abstraction

La fonction d'abstraction est absolument identique à la fonction d'abstraction relative au domaine précédent (cf. définition 4.6).

5.3 Préordre

On opère sur la définition du préordre les mêmes modifications que sur la définition de la fonction de concrétisation.

Définition 5.3 ($a \overset{f}{\leftarrow} a'$)

Soient a et a' appartenant à $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
On dira que a' approxime a au travers de f, i.e. $a \overset{f}{\leftarrow} a'$, si et seulement si a, a' et f vérifient les trois propriétés ci-dessous.

1. $f : dom(a'_1) \rightarrowtail dom(a_1)$
2. $a_0 = \bot_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $a'_0 = \bot_0^{\#}[v_1/t_1\alpha'_1, ..., v_n/t_n\alpha'_n, \mathbf{this}/t_{n+1}\alpha'_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : f(\alpha'_i) = \alpha_i$
3. $\forall l \in dom(f)$,

$$a'_1(l) = t \in Type \Rightarrow a_1(f(l)) = t$$

$$a'_1(l) = e \in Type^{\#} \Rightarrow \begin{cases} \textbf{soit } a_1(f(l)) \in Type \wedge a_1(f(l)) \in \gamma(e) \\ \textbf{soit } a_1(f(l)) \in Type^{\#} \wedge a_1(f(l)) \leq e \\ \textbf{soit } a_1(f(l)) = (nc, v) \in \mathbb{I}nst^{\#} \wedge nc \in \gamma(e) \end{cases}$$

$$a'_1(l) = (nc, v') \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} a_1(f(l)) = (nc, v) \in \mathbb{I}nst^{\#} \\ dom(v) = dom(v') \\ \forall ch \in dom(v),\ f(v'(ch)) = v(ch) \end{cases}$$

La figure 5.1 donne un exemple d'approximations successives : la première situation fournit une information "complète" tant au niveau des types qu'au niveau du "sharing"; cette première situation est approximée au travers de la fonction f par la deuxième situation qui conserve une information complète au niveau des types mais ne permet plus d'affirmer que les deux variables désignent la même instance; cette deuxième situation est elle-même approximée par la troisième situation au travers de la fonction f'; dans cette dernière situation, on sait uniquement que x et y sont de type A.

Proposition 5.2 (Domaine de la fonction de mise en correspondance)

Soient a et a' appartenant à $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$.
Si $a \overset{f}{\leftarrow} a'$, on a l'implication suivante.
$a' \in \mathbb{D}^{\#} \Rightarrow f$ *partout définie*

Définition 5.4 (Préordre sur $\mathbb{D}^{\#}$)

Soient deux éléments a et a' de $\mathbb{D}^{\#}$. On dira que a' approxime a (ce qu'on note $a \leq^{*2} a'$) si et seulement si il existe une fonction f telle que $a \overset{f}{\leftarrow} a'$.

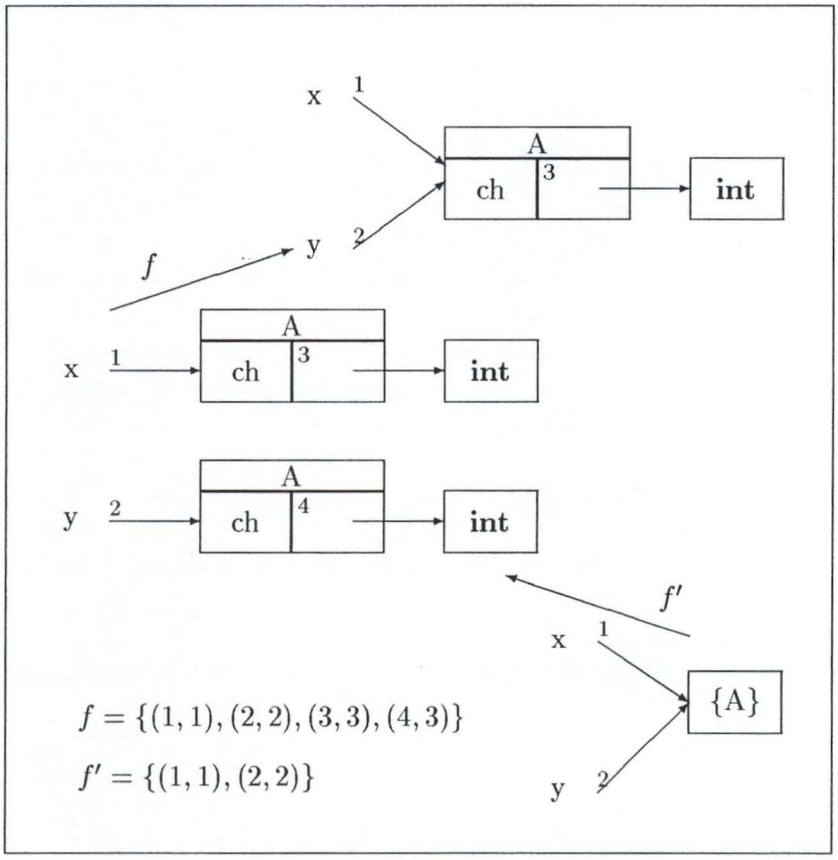

Figure 5.1: Exemples d'approximations

Proposition 5.3 (Relation de préordre)

La relation $\leq^{*2}$ définie sur $\mathbb{D}^{\#}$ est effectivement une relation de préordre, i.e. elle est réflexive et transitive.

Preuve :

Il est évident qu'on a bien $a \xleftarrow{id} a$

On prouve facilement $a \xleftarrow{f \circ g} c$ à partir de $a \xleftarrow{f} b$ et $b \xleftarrow{g} c$ (une preuve analogue est fournie pour la proposition suivante). $\Diamond$

Proposition 5.4 (Cohérence de $\leq^{*2}$ par rapport à γ^{*2})

Soient a et a' appartenant à $\mathbb{D}^{\#}$. Si a' approxime a, la concrétisation de a' contient la concrétisation de a, i.e.

$$a \leq^{*2} a' \Rightarrow \gamma^{*2}(a) \subseteq \gamma^{*2}(a').$$

Preuve :

Cette preuve est évidement fort proche de son homologue présentée au chapitre 4 si ce n'est qu'on "renverse le sens des flèches".

Soit $d \in \gamma^{*2}(a)$. Montrons que $d \in \gamma^{*2}(a')$.

Puisque $d \in \gamma^{*2}(a)$, il existe une fonction f telle que $d \xleftarrow{f} a$. D'autre part, puisque $a \leq^{*2} a'$, il existe une fonction g telle que $a \xleftarrow{g} a'$.

Si la composée de g et f vérifie $d \xleftarrow{f \circ g} a'$, on a bien que d est un élément de $\gamma^{*2}(a')$.

1. $f \circ g : dom(a'_1) \nrightarrow dom(d_1)$
2. $d_0 = \perp_0[v_1/t_1\delta_1, ..., v_n/t_n\delta_n, \mathbf{this}/t_{n+1}\delta_{n+1}]$
   $a_0 = \perp_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $a'_0 = \perp_0^{\#}[v_1/t_1\alpha'_1, ..., v_n/t_n\alpha'_n, \mathbf{this}/t_{n+1}\alpha'_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : f \circ g(\alpha'_i) = f(g(\alpha'_i)) = f(\alpha_i) = \delta_i$
3. Soit $l \in dom(f \circ g)$.
   $a'(l) = t \in Type \Rightarrow a(g(l)) = t \Rightarrow d(f(g(l))) = t$

   La première implication découle du fait que $a \xleftarrow{g} a'$ et la seconde du fait que $d \xleftarrow{f} a$.

De même, puisque $a \overset{g}{\leftarrow} a'$,

$$a'_1(l) = e \in Type^{\#} \Rightarrow \begin{cases} \textbf{soit } a_1(g(l)) \in Type \wedge a_1(g(l)) \in \gamma(e) \\ \textbf{soit } a_1(g(l)) \in Type^{\#} \wedge a_1(g(l)) \leq e \\ \textbf{soit } a_1(g(l)) = (nc, va) \in \mathbb{I}nst^{\#} \wedge nc \in \gamma(e) \end{cases}$$

et puisque $d \overset{f}{\leftarrow} a$,

$a_1(g(l)) \in Type \Rightarrow d_1(f(g(l))) \in \mathbb{B}ase \wedge p_1(d_1(f(g(l)))) = a_1(g(l))$
$a_1(g(l)) \in Type^{\#} \Rightarrow p_1(d_1(f(g(l)))) \in \gamma(a_1(g(l)))$

$$a_1(g(l)) = (nc, va) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} d_1(f(g(l))) = (nc, v) \in \mathbb{I}nst \\ dom(v) = dom(va) \\ \forall ch \in dom(v),\ f(va(ch)) = v(ch) \end{cases} \tag{5.1}$$

On obtient donc

$$a'_1(l) = e \in Type^{\#} \Rightarrow \begin{cases} \textbf{soit } d_1(f(g(l))) \in \mathbb{B}ase \wedge p_1(d_1(f(g(l)))) = a_1(g(l)) \in \gamma(e) \\ \textbf{soit } a_1(g(l)) \in Type^{\#} \wedge p_1(d_1(f(g(l)))) \in \gamma(a_1(g(l))) \subseteq \gamma(e) \\ \qquad \text{(puisque } \gamma \text{ est monotone par rapport à } \leq) \\ \textbf{soit } d_1(f(g(l))) = (nc, v) \in \mathbb{I}nst \wedge nc \in \gamma(e) \end{cases}$$

En bref, on a donc ce qu'on souhaite :
$a'_1(l) = e \in Type^{\#} \Rightarrow p_1(d_1(f(g(l)))) \in \gamma(e)$.

Passons enfin au dernier cas de figure.

$$a'_1(l) = (nc, va') \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} a_1(g(l)) = (nc, va) \in \mathbb{I}nst^{\#} \\ dom(va) = dom(va') \\ \forall ch \in dom(va),\ g(va'(ch)) = va(ch) \end{cases}$$

Combinée avec l'implication (5.1), l'implication ci-dessus fournit :

$$a'_1(l) = (nc, va') \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} d_1(f(g(l))) = (nc, v) \in \mathbb{I}nst \\ dom(va) = dom(va') = dom(v) \\ \forall ch \in dom(v),\ v(ch) = f(va(ch)) = f(g(va'(ch)) \end{cases}$$

Et la troisième condition est vérifiée.

◊

5.4 Borne supérieure

Nous commençons dans cette section par donner quelques exemples de borne supérieure. Nous expliquons ensuite un algorithme pour le calcul de celle-ci et nous terminons par la présentation de la preuve de correction de cet algorithme.

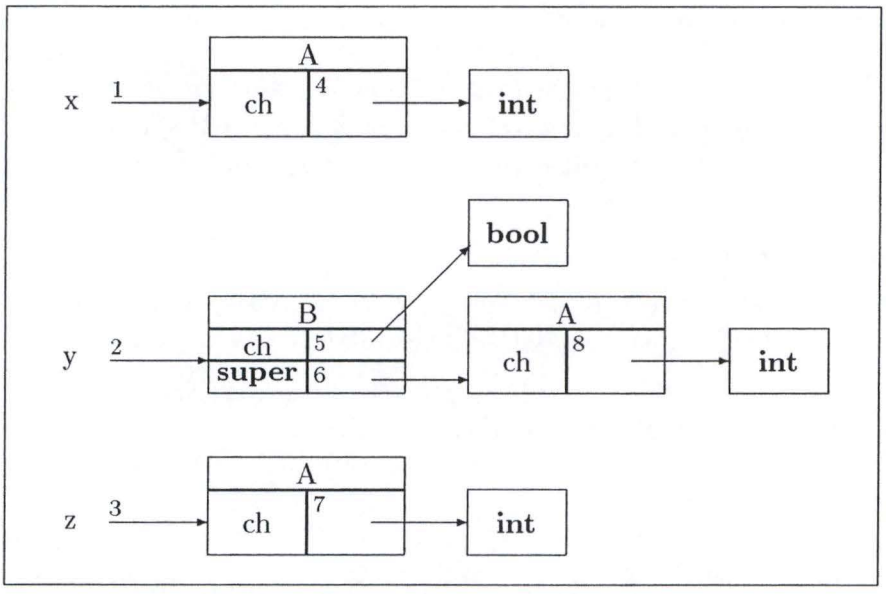

Figure 5.2: Situation $S1$

5.4.1 Quelques exemples

Nous choisissons de reprendre les exemples présentés au chapitre 4. Soulignons cependant que les éléments du domaine $\mathbb{D}^{\#}$ considérés n'ont absolument plus le même sens en termes de situations concrètes approximées par ceux-ci.

Dans la situation $S1$ (cf. figure 5.2), on a aucune information sur le "sharing" tandis que, dans la situation $S2$ (cf. figure 5.3), on sait que x, z et y.**super** désignent la même instance. Puisque $S1$ et $S2$ donnent les mêmes informations de type, il est évident que la borne supérieure de ces deux situations ne peut être qu'isomorphique à $S1$. D'un autre côté, on vérife facilement qu'il existe f telle que $S2 \overset{f}{\leftarrow} S1$.

Considérons maintenant une troisième situation $S3$ (cf. figure 5.4) toujours relative au même environnement. Dans cette situation le type dynamique de x change et on sait que, d'une part x et y désignent la même instance et que, d'autre part, y.**super** et z désignent également la même instance.

Intéressons nous à la borne supérieure de $S2$ et $S3$. Puisque dans ces deux situations x a un type différent, on ne peut conserver qu'une approximation de celui-ci. y.**super** et z désignent la même instance à la fois dans $S2$ et dans $S3$ (et les types de ces instances sont évidemment identiques dans les deux situations); conserver une information "maximale" impose donc de conserver ce point de partage. On aboutit par conséquent à la borne supérieure représentée dans la figure 5.5.

Attaquons-nous à un dernier exemple. Les situations $S4$ et $S5$, symbolisées respectivement par les figures 5.6 et 5.7, manipulent deux variables de type B. Le type B possède un seul champ

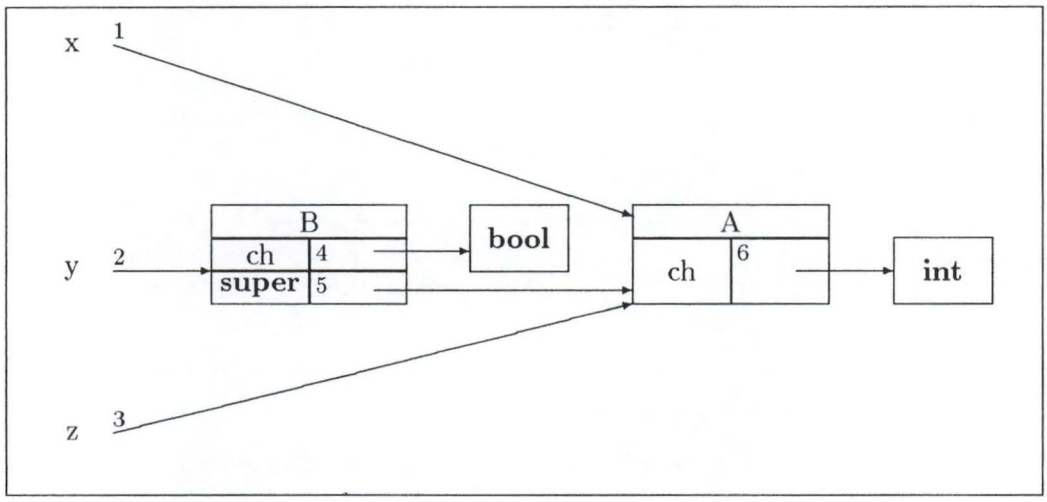

Figure 5.3: Situation $S2$

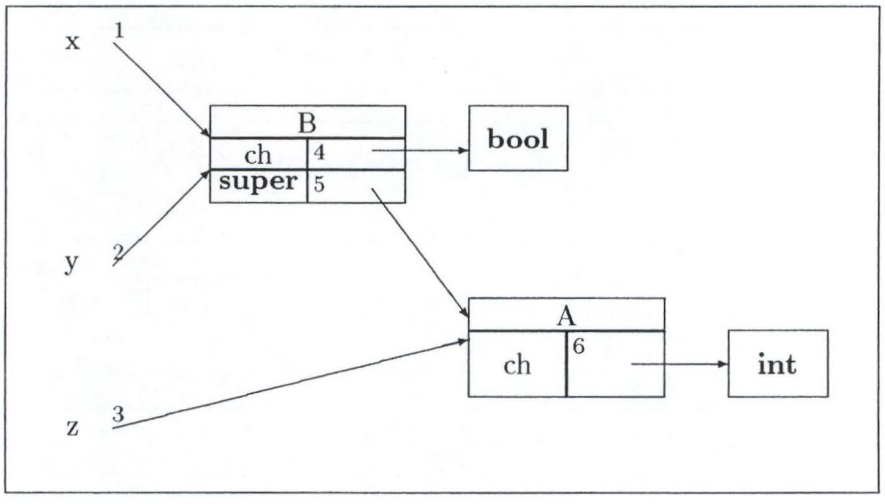

Figure 5.4: Situation $S3$

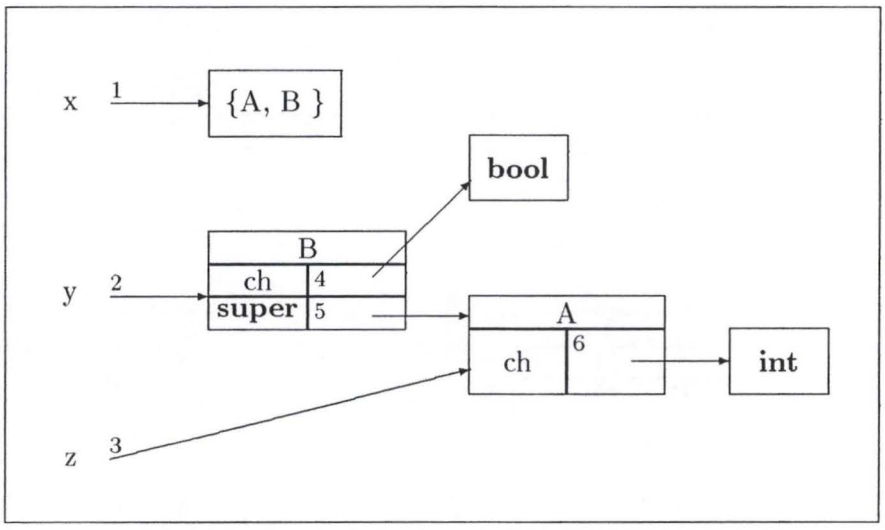

Figure 5.5: Borne supérieure de $S2$ et $S3$

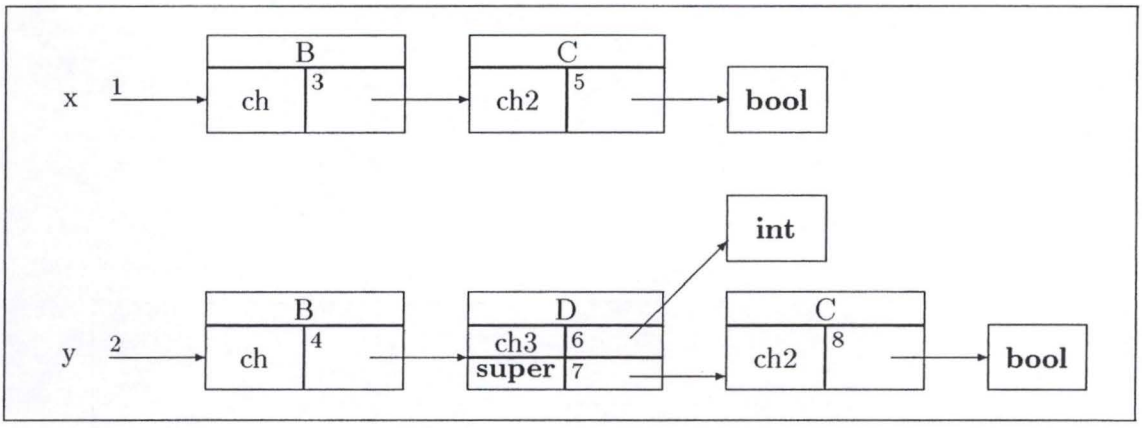

Figure 5.6: Situation $S4$

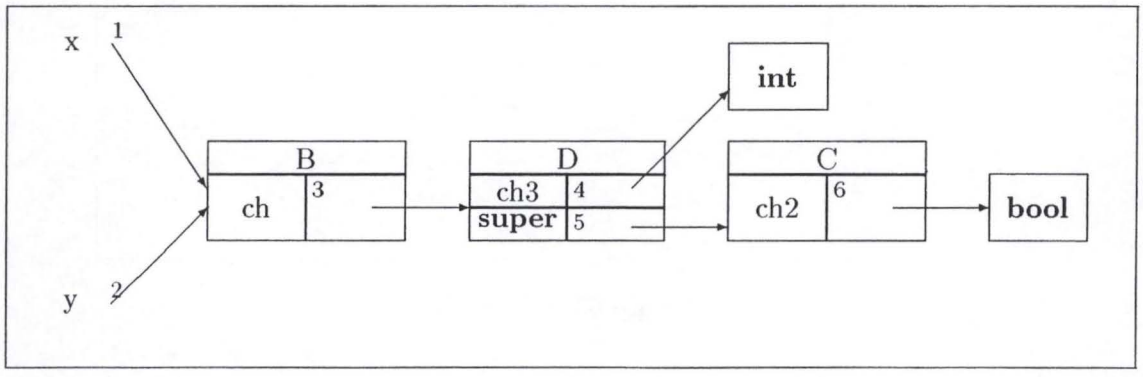

Figure 5.7: Situation $S5$

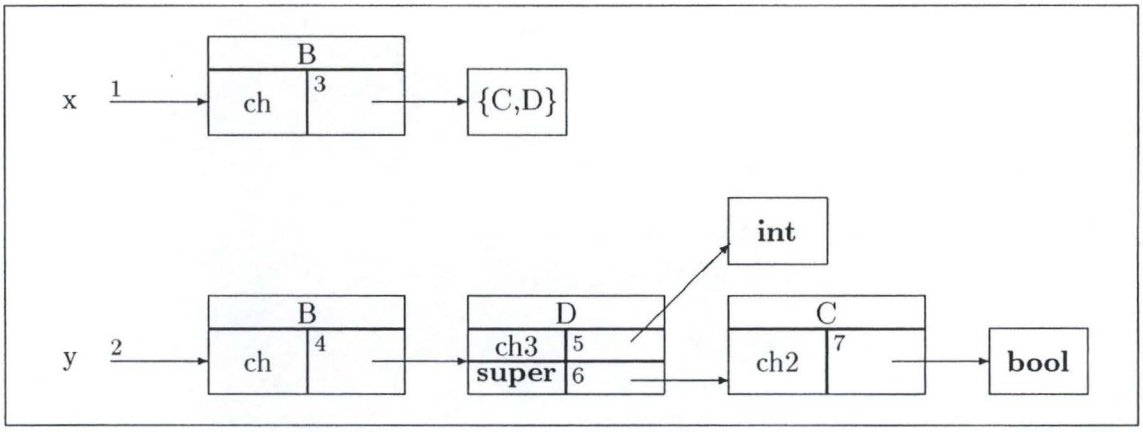

Figure 5.8: Borne supérieure de $S4$ et $S5$: $S6$

ch déclaré de type C. Ce dernier type peut être étendu par le type D. Dans la situation $S4$, on ne dispose d'aucune information relative au "sharing" tandis que dans la situation $S5$ on sait que x et y désignent la même instance. Il est évident qu'on ne peut que perdre cette dernière information.

Les types de x et des y sont identiques dans les deux situations, il est donc logique que la borne supérieure conserve une information complète à ce sujet. Dans $S4$, le type de $x.ch$ est C tandis que, dans $S5$, ce type est D. La borne supérieure doit donc approximer le type de ce désignateur. Il n'en va pas de même pour le champ *ch* de y qui est de type D dans les deux cas. En regroupant ces différentes considérations, on obtient la borne supérieure fournie par la figure 5.8.

Maintenant que ces quelques exemples nous ont permis de développer notre intuition de la borne supérieure, nous pouvons passer à la construction d'un algorithme permettant le calcul de celle-ci.

5.4.2 Quelques notations

Avant d'entamer la présentation de l'algorithme de calcul de la borne supérieure, précisons quelques notations utiles à l'expression de celui-ci ou à l'expression de sa preuve de correction.

La fonction $\tilde{\alpha}$ étend juste la portée de la fonction primitive d'abstraction α.

$$\begin{array}{rccll} \tilde{\alpha}: & \mathbb{V}al^{\#} & \longrightarrow & Type^{\#} & \\ & e & \rightsquigarrow & e & \text{si } e \in Type^{\#} \\ & t & \rightsquigarrow & \alpha(t) & \text{si } t \in Type \\ & (nc, v) & \rightsquigarrow & \alpha(nc) & \text{si } (nc, v) \in \mathbb{I}nst^{\#} \end{array}$$

Soient a appartenant à $\mathbb{D}^{\#}$, b appartenant $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$ et W un ensemble de locations abstraites.

On emploiera la notation $a \overset{f}{\leftarrow} (b, W)$ si et seulement si f, a, b et W vérifient les quatre conditions

suivantes :

1. $W \cap dom(b_1) = \emptyset$
2. $f : dom(b_1) + W \longrightarrow dom(a_1)$
3. $a_0 = \bot_0^{\#}[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, \mathbf{this}/t_{n+1}\alpha_{n+1}]$
   $b_0 = \bot_0^{\#}[v_1/t_1\beta_1, ..., v_n/t_n\beta_n, \mathbf{this}/t_{n+1}\beta_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : f(\beta_i) = \alpha_i$
4. $\forall l \in dom(b_1)$,

$$b_1(l) = t \in Type \Rightarrow a_1(f(l)) = t$$

$$b_1(l) = e \in Type^{\#} \Rightarrow \begin{cases} \mathbf{soit}\ a_1(f(l)) \in Type \wedge a_1(f(l)) \in \gamma(e) \\ \mathbf{soit}\ a_1(f(l)) \in Type^{\#} \wedge a_1(f(l)) \leq e \\ \mathbf{soit}\ a_1(f(l)) = (nc, v) \in \mathbb{I}nst^{\#} \wedge nc \in \gamma(e) \end{cases}$$

$$b_1(l) = (nc, vb) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} a_1(f(l)) = (nc, va) \in \mathbb{I}nst^{\#} \\ dom(va) = dom(vb) \\ \forall ch \in dom(vb),\ f(vb(ch)) = va(ch) \end{cases}$$

Il est clair que le concept couvert par cette notation est une généralisation de l'approximation notée $a \overset{f}{\leftarrow} b$: si $W = \emptyset$, $a \overset{f}{\leftarrow} b$ si et seulement si $a \overset{f}{\leftarrow} (b, W)$.

Il s'agit ici d'exprimer que b est un majorant "en cours de construction" pour a. Les locations contenues dans l'ensemble W coïncident avec les points pour lesquels la borne supérieure "n'a pas encore été calculée" mais qui doivent intervenir dans celle-ci "au vu" de b.

On introduit la généralisation "inverse". Soit b un élément de $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$ et c un élément de $\mathbb{D}^{\#}$ ainsi que W un ensemble de locations abstraites.

On emploiera la notation $(b, W) \overset{h}{\leftarrow} c$, si et seulement si h, b, c et W vérifient les quatre conditions suivantes :

1. $W \cap dom(b_1) = \emptyset$
2. $h : dom(c_1) \nrightarrow dom(b_1) + W$
3. $c_0 = \bot_0^{\#}[v_1/t_1\gamma_1, ..., v_n/t_n\gamma_n, \mathbf{this}/t_{n+1}\gamma_{n+1}]$
   $b_0 = \bot_0^{\#}[v_1/t_1\beta_1, ..., v_n/t_n\beta_n, \mathbf{this}/t_{n+1}\beta_{n+1}]$

   $\forall i : 1 \leq i \leq n+1 : h(\gamma_i) = \beta_i$

4. $\forall l \in dom(h) \wedge h(l) \notin W$,

$$c_1(l) = t \in Type \Rightarrow b_1(h(l)) = t$$

$$c_1(l) = e \in Type^{\#} \Rightarrow \begin{cases} \textbf{soit } b_1(h(l)) \in Type \wedge b_1(h(l)) \in \gamma(e) \\ \textbf{soit } b_1(h(l)) \in Type^{\#} \wedge b_1(h(l)) \leq e \\ \textbf{soit } b_1(h(l)) = (nc, v) \in \mathbb{I}nst^{\#} \wedge nc \in \gamma(e) \end{cases}$$

$$c_1(l) = (nc, vc) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} b_1(h(l)) = (nc, vb) \in \mathbb{I}nst^{\#} \\ dom(vc) = dom(vb) \\ \forall ch \in dom(vb),\ h(vc(ch)) = vb(ch) \end{cases}$$

De nouveau, si $W = \emptyset$, $(b, W) \overset{h}{\leftarrow} c$ est équivalent à $b \overset{h}{\leftarrow} c$.

Cette notation exprime que c approxime b au travers de h; b étant "en cours de construction". Les propriétés d'approximation ne doivent pas être vérifiées pour les éléments de W puisque cet ensemble correspond à la partie "inachevée" de b.

5.4.3 Algorithme

On propose dans cette section un algorithme permettant de calculer la borne supérieure de deux éléments de $(\mathbb{D}^{\#}, \leq^{*2})$.

Plus précisément, nous proposons un algorithme vérifiant la spécification suivante.

Objets utilisés :

a, a' : $\mathbb{D}^{\#}$
b : $\mathbb{E}nv^{\#} \times \mathbb{S}tore^{\#}$
f, f' : $\mathbb{L}oc^{\#} \rightarrowtail \mathbb{L}oc^{\#}$

Précondition :

$a_0 = \bot_0[v_1/t_1\alpha_1, ..., v_n/t_n\alpha_n, v_{n+1}/t_{n+1}\alpha_{n+1}]$
$a'_0 = \bot_0[v_1/t_1\alpha'_1, ..., v_n/t_n\alpha'_n, v_{n+1}/t_{n+1}\alpha'_{n+1}]$

Postcondition :

$a \overset{f}{\leftarrow} b \wedge a' \overset{f'}{\leftarrow} b$
$b \in \mathbb{D}^{\#}$
$\forall c, g, g' : c \in \mathbb{D}^{\#} \wedge a \overset{g}{\leftarrow} c \wedge a' \overset{g'}{\leftarrow} c : \exists h,\ b \overset{h}{\leftarrow} c$

Il est évident que la postcondition ci-dessus pourrait aussi bien s'exprimer directement en termes de la relation de préordre $\leq^{*2}$, mais nous avons opté pour une expression manipulant explicitement les fonctions de mise en correspondance.

L'algorithme proposé est repris dans la figure 5.9.

Cet algorithme repose sur une construction pas à pas de la borne supérieure b et des fonctions d'approximations f et f'. Pour ce faire, on initialise les fonctions d'approximations et b de

begin
$b_0 := \bot_0[v_1/t_1\beta_1, ..., v_n/t_n\beta_n, v_{n+1}/t_{n+1}\beta_{n+1}]$
$b_1 := \bot_1$
$f := \{(\beta_1, \alpha_1), ..., (\beta_{n+1}, \alpha_{n+1})\}$
$f' := \{(\beta_1, \alpha'_1), ..., (\beta_{n+1}, \alpha'_{n+1})\}$
$W := \{\beta_1, ..., \beta_{n+1}\}$
while $W \neq \emptyset$ **do**
  begin
    $W \rightarrow W + \{\beta\}$
    if $a_1(f(\beta)) = a'_1(f'(\beta)) = t \in Type$
    then $b_1(\beta) := t$
    else
      if $a_1(f(\beta)) = (nc, va) \in \mathbb{I}nst^{\#} \wedge a'_1(f'(\beta)) = (nc, va') \in \mathbb{I}nst^{\#}$
        $\wedge\, dom(va) = \{x_1, ..., x_m\}$
      then
        begin
          $b_1(\beta) := (nc, v)$
          if $\exists(l_1, ..., l_m), \forall i : 1 \leq i \leq m : f(l_i) = va(x_i) \wedge f'(l_i) = va'(x_i)$
          then
            for-each $i \in \{1, ..., m\}$ **do** $v(x_i) := ll_i$
              where $\forall i : 1 \leq i \leq m : f(ll_i) = va(x_i) \wedge f'(ll_i) = va'(x_i)$
          else
            begin
              for-each $i \in \{1, ..., m\}$ **do** $v(x_i) := ll_i$
                where $\{ll_1, ..., ll_m\} \cap (dom(b_1) + W) = \emptyset$
              $f := f + \{(ll_1, va(x_1)), ..., (ll_m, va(x_m))\}$
              $f' := f' + \{(ll_1, va'(x_1)), ..., (ll_m, va'(x_m))\}$
              $W := W + \{ll_1, ..., ll_m\}$
            end
        end
      else $b_1(\beta) := \tilde{\alpha}(a_1(f(\beta))) \sqcup \tilde{\alpha}(a'_1(f'(\beta)))$
  end
end

Figure 5.9: Calcul de $a \sqcup a'$.

manière à faire correspondre les différents environnements. Le domaine du store de b est alors vide et les locations de l'environnement de celui-ci consituent les éléments "restant à traiter".

A partir de là, on traite à chaque itération un élément de W (correspondant à un point de la borne supérieure "pas encore construit"). Ce point détermine une location dans a et une location "correspondante" dans a'. Si les deux stores donnent en ces points des informations "non compatibles", on procède à une approximation (la plus fine possible). Si les deux stores renvoient le même élément de type, on conserve simplement ce type. Si les deux stores renvoient une instance du même type, on doit "conserver" cette instance : si on a déjà construit une instance correspondant à la fois à l'instance renvoyée par a et à l'instance renvoyée par a', on conserve simplement cette instance (conservation maximale du sharing), sinon on crée une nouvelle instance.

En fait, la construction de cet algorithme se base sur les propriétés invariantes ci-après. Ces dernières s'expriment en fonction d'un élément arbitraire c de $\mathbb{D}^\#$ tel que $a \overset{g}{\leftarrow} c$ et $a' \overset{g'}{\leftarrow} c$ (il s'agit là d'une simple traduction d'une quantification universelle sur c).

1. On construit un majorant de a et a'.

$$a \overset{f}{\leftarrow} (b, W) \wedge a' \overset{f'}{\leftarrow} (b, W) \qquad \text{(I1)}$$

2. On construit effectivement une borne supérieure.

$$\beta \neq \beta' \in dom(b_1) + W \Rightarrow \begin{cases} f(\beta) = f(\beta') \Rightarrow f'(\beta) \neq f'(\beta') \\ f'(\beta) = f'(\beta') \Rightarrow f(\beta) \neq f(\beta') \end{cases} \qquad \text{(I2)}$$

La fonction h définie par

$$\begin{array}{l} h : dom(c_1) \rightharpoonup dom(b_1) + W \\ dom(h) = \{l \in dom(c_1) \mid \exists \beta \in dom(b_1) + W \wedge f(\beta) = g(l) \wedge f'(\beta) = g'(l)\} \\ \forall l \in dom(h),\ h(l) \in f^{\leftarrow}(g(l)) \cap f'^{\leftarrow}(g'(l)) \end{array}$$

vérifie

$$(b, W) \overset{h}{\leftarrow} c. \qquad \text{(I3)}$$

3. On construit bien un élément de $\mathbb{D}^\#$.

$$dom(b_1) + W = \{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} Attloc\ b_1\ \beta_i \qquad \text{(I4)}$$

Remarquons que la propriété (I2) implique que l'ensemble $f^{\leftarrow}(g(l)) \cap f'^{\leftarrow}(g'(l))$ comprend au plus un élément et que, par conséquent, la fonction h est bien complètement définie.

Soulignons également, dés à présent, que nous ne fournissons qu'une preuve partielle de cor-

rection : nous ne prouvons pas complètement que b appartient à $\mathbb{D}^{\#}$ puisque nous ne nous attardons que sur la propriété de "coupure du store".

5.4.4 Correction de l'algorithme

Cette preuve se réalise, classiquement, en quatre étapes :

1. on prouve que les propriétés invariantes sont vérifiées après l'initialisation;
2. on prouve que le programme se termine;
3. on prouve que si $W = \emptyset$, les propriétés invariantes impliquent la postcondition;
4. on prouve que l'itération conserve les propriétés invariantes.

Vérification des propriétés à l'initialisation

(I1) est trivialement vérifiée puisque $dom(b_1) = \emptyset$ et que

$$\forall i : 1 \leq i \leq n+1 : f(\beta_i) = \alpha_i \wedge f'(\beta_i) = \alpha'_i.$$

(I2) est également vérifiée puisque

$$\forall \beta, \beta' \in dom(b_1) + W, \beta \neq \beta' \Rightarrow \begin{cases} f(\beta) \neq f(\beta') \\ f'(\beta) \neq f'(\beta') \end{cases}$$

(I3) On peut montrer que

$$h = \{(\gamma_1, \beta_1), ..., (\gamma_{n+1}, \beta_{n+1})\}$$

(où $\{\gamma_1, ..., \gamma_{n+1}\}$ correspond à l'ensemble des locations initiales de c_1)

or

$$\forall i : 1 \leq i \leq n+1 : \beta_i \in W$$

Par conséquent, on a immédiatement $(b, W) \overset{h}{\leftarrow} c$.

La propriété (I4) est également aisément prouvée : d'une part,

$$dom(b_1) + W = W = \{\beta_1, ...\beta_{n+1}\},$$

et d'autre part,

$$\{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ b_1\ \beta_i = \{\beta_1, ..., \beta_{n+1}\} \cup \emptyset.$$

Terminaison

On peut déduire de (I1) + (I2) l'inégalité suivante :

$$\#(dom(b_1)) + W \leq (\#dom(a_1)) \times (\#dom(a'_1))$$

donc, a fortiori,

$$\#(dom(b_1)) \leq (\#dom(a_1)) \times (\#dom(a'_1))$$

Or il est évident que le nombre d'éléments de l'ensemble $dom(b_1)$ croît strictement à chaque itération.

Postcondition

Puisque $W = \emptyset$, il est clair que (I1) se réécrit

$$a \overset{f}{\leftarrow} b \wedge a \overset{f'}{\leftarrow} b.$$

De même, (I3) se réécrit $b \overset{h}{\leftarrow} c$.

La propriété (I4) se réduit quant à elle à l'équation

$$dom(b_1) = \{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ b_1\ \beta_i$$

qui est une simple réécriture de la propriété de "coupure du store".

Itération

Remarque : lorsque ce sera nécessaire, nous indicerons les valeurs des différentes variables avant l'itération par un zéro (et seulement dans ces cas-là).

Cas 1 : $a_1(f(\beta)) = a'_1(f'(\beta)) = t \in Type$

Dans ce cas, les seules modifications sont

$$W = W_0 \setminus \{\beta\}$$
$$b_1 = (b_1)_0[\beta/t]$$

(I1) :
La seule partie de la propriété susceptible d'être modifiée est la condition 4 de la définition de $a \overset{f}{\leftarrow} (b, W)$ (respectivement $a' \overset{f'}{\leftarrow} (b, W)$) puisque le domaine de b_1 est augmenté de l'élément β.

Or, on a bien,

$$b_1(\beta) = t \in Type \Rightarrow a_1(f(\beta)) = t.$$

(I2) :
Puisque les fonctions sont inchangées, cette propriété reste vérifiée.

(I3) :
De nouveau la seule partie de propriété susceptible d'être modifiée se situe au niveau de la condition 4 de la définition de $(b, W) \overset{h}{\leftarrow} c$ puisqu'il peut exister l tel que $h(l) = \beta$ et que $\beta \notin W$.

Si $h(l)$ est définie alors $\{h(l)\} = f^{\leftarrow}(g(l)) \cap f'^{\leftarrow}(g'(l))$.

Par conséquent,

$$a_1(f(\beta)) = a_1(g(l)) = t$$
$$a'_1(f'(\beta)) = a'_1(g'(l)) = t$$

et donc, puisque $a \overset{g}{\leftarrow} c$ et $a' \overset{g'}{\leftarrow} c$, on ne peut avoir que les deux situations suivantes :

soit $c_1(l) = t$
soit $c_1(l) = e \wedge t \in \gamma(e)$

Finalement,

soit $c_1(l) = t \in Type \wedge b_1(h(l)) = b_1(\beta) = t$
soit $c_1(l) = e \in Type^{\#} \wedge b_1(h(l)) = t \wedge t \in \gamma(e)$

(I4) :
La conservation de cette propriété est évidente puisqu'on a les égalités suivantes.

$$dom(b_1) + W = (dom(b_1)_0 + \{\beta\}) + W_0 \setminus \{\beta\} = dom(b_1)_0 + W_0$$
$$\{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ b_1\ \beta_i = \{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ (b_1)_0\ \beta_i$$

Cas 2 :

$$a_1(f(\beta)) = (nv, va) \in \mathbb{I}nst^{\#} \wedge a'_1(f'(\beta)) = (nc, va') \in \mathbb{I}nst^{\#}$$
$$dom(va) = dom(va') = \{x_1, ..., x_m\}$$
$$\exists (l_1, ..., l_m), \forall i : 1 \leq i \leq m : f(l_i) = va(x_i) \wedge f'(l_i) = va'(x_i)$$

Sans perdre de généralité, nous supposerons que $dom(va) = \{x\}$.

Dans ce cas, les seules modifications sont

$$W = W_0 \setminus \{\beta\}$$
$$b_1 = (b_1)_0[\beta/(nc, v)]$$
$$\text{avec } dom(v) = \{x\} \wedge v(x) = ll \wedge f(ll) = va(x) \wedge f'(ll) = va'(x)$$

(I1) :
La seule partie de la propriété susceptible d'être modifiée est la condition 4 de la définition de $a \overset{f}{\leftarrow} (b, W)$ (respectivement $a' \overset{f'}{\leftarrow} (b, W)$) puisque le domaine de b_1 est augmenté de l'élément β.

Or, on a bien,

$$b_1(\beta) = (nc, v) \in \mathbb{I}nst^{\#} \Rightarrow \begin{cases} a_1(f(\beta)) = (nc, va) \in \mathbb{I}nst^{\#} \\ dom(va) = \{x\} = dom(v) \\ va(x) = f(ll) = f(v(x)) \end{cases}$$

(I2) :
Puisque les fonctions sont inchangées, cette propriété reste vérifiée.

(I3) :
De nouveau la seule partie de propriété susceptible d'être modifiée se situe au niveau de la condition 4 de la définition de $(b, W) \overset{h}{\leftarrow} c$ puisqu'il peut exister l tel que $h(l) = \beta$ et que $\beta \notin W$.

Si $h(l)$ est définie alors $\{h(l)\} = f^{\leftarrow}(g(l)) \cap f'^{\leftarrow}(g'(l))$.

Par conséquent,

$$a_1(f(\beta)) = a_1(g(l)) = (nc, va)$$
$$a'_1(f'(\beta)) = a'_1(g'(l)) = (nc, va')$$

et donc, puisque $a \overset{g}{\leftarrow} c$ et $a' \overset{g'}{\leftarrow} c$, on ne peut avoir que les deux situations suivantes :

$$\textbf{soit} \begin{cases} c_1(l) = (nc, vc) \in \mathbb{I}nst^{\#} \\ dom(vc) = dom(va) = dom(va') \\ va(x) = g(vc(x)) \\ va'(x) = g'(vc(x)) \end{cases}$$
$$\textbf{soit } c_1(l) = e \wedge nc \in \gamma(e)$$

Dans le premier cas,

$$vc(x) \in dom(c_1) \wedge f(ll) = va(x) = g(vc(x)) \wedge f'(ll) = va'(x) = g'(vc(x))$$

et donc $vc(x)$ appartient au domaine de h et $v(x) = h(vc(x))$ par définition de h.

Finalement,

$$\textbf{soit } c_1(l) = (nc, vc) \in Type^{\#} \wedge \begin{cases} b_1(h(l)) = b_1(\beta) = (nc, v) \\ dom(v) = dom(vc) = \{x\} \\ v(x) = h(vc(x)) \end{cases}$$
$$\textbf{soit } c_1(l) = e \in Type^{\#} \wedge b_1(h(l)) = (nc, v) \wedge nc \in \gamma(e)$$

(I4) :
La conservation de cette propriété est évidente puisqu'on a toujours les égalités suivantes.

$$dom(b_1) + W = (dom(b_1)_0 + \{\beta\}) + W_0 \setminus \{\beta\} = dom(b_1)_0 + W_0$$
$$\{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ b_1\ \beta_i = \{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ (b_1)_0\ \beta_i$$

Cas 3 :

$$a_1(f(\beta)) = (nv, va) \in \mathbb{I}nst^{\#} \wedge a'_1(f'(\beta)) = (nc, va') \in \mathbb{I}nst^{\#}$$
$$dom(va) = dom(va') = \{x\}$$
$$\neg(\exists l, f(l) = va(x) \wedge f'(l) = va'(x))$$

Dans ce cas, les modifications effectuées sont

$$W = (W_0 \setminus \{\beta\}) + \{ll\}$$
$$b_1 = (b_1)_0[\beta/(nc, v)]$$
avec $dom(v) = \{x\} \wedge v(x) = ll$
$$f = f_0 + \{(ll, va(x))\}$$
$$f' = f'_0 + \{(ll, va'(x))\}$$

Les preuves de conservation des propriétés (I1) et (I3) sont très similaires à celles présentées pour le cas précédent. La propriété intéressante est ici (I2).

(I2) :
Soient β et β' appartenant à $dom(b_1) + W$ tels que $\beta \neq \beta'$.

Si $\beta \neq ll$ et $\beta' \neq ll$, on applique simplement la propriété avant l'itération.

Considérons donc, sans perdre de généralité, $\beta = ll$ et $\beta' \in dom(b_1)_0 + W_0$.

Supposons par l'absurde que

$$f(\beta) = f(ll) \wedge f'(\beta) = f'(ll).$$

On a alors

$$f(\beta) = f(ll) = va(x) \wedge f'(\beta) = f'(ll) = va'(x).$$

D'autre part,

$$f(\beta) = f_0(\beta) \wedge f'(\beta) = f'_0(\beta).$$

Par conséquent,

$$f_0(\beta) = va(x) \wedge f'_0(\beta) = va'(x).$$

Ce qui est impossible puisque

$$\neg(\exists l, f_0(l) = va(x) \wedge f'_0(l) = va'(x)).$$

(I4) :
D'un côté, on a

$$dom(b_1) + W = (dom(b_1)_0 + \{\beta\}) + W_0 \setminus \{\beta\} + \{ll\} = (dom(b_1)_0 + W_0) + \{ll\}$$

et de l'autre

$$\{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ b_1\ \beta_i = \{\beta_1, ..., \beta_{n+1}\} \cup \bigcup_{i=1}^{n+1} \mathcal{A}ttloc\ (b_1)_0\ \beta_i + \{ll\}$$

Cas 4 : (exemple)

$$a_1(f(\beta)) = e \in Type^{\#} \wedge a'_1(f'(\beta)) = e' \in Type^{\#}$$

Dans ce cas, les seules modifications sont

$$W = W_0 \setminus \{\beta\}$$
$$b_1 = (b_1)_0[\beta / e \sqcup e')]$$

On montre juste la conservation des propriétés (I1) et (I3).

(I1) :
La seule partie de la propriété susceptible d'être modifiée est la condition 4 de la définition de $a \xleftarrow{f} (b, W)$ (respectivement $a' \xleftarrow{f'} (b, W)$) puisque le domaine de b_1 est augmenté de l'élément β.

Or on a bien,

$$b_1(\beta) = e \sqcup e' \in Type^{\#} \wedge a_1(f(\beta)) = e \leq e \sqcup e'$$

(I3) :
De nouveau la seule partie de propriété susceptible d'être modifiée se situe au niveau de la condition 4 de la définition de $(b, W) \xleftarrow{h} c$ puisqu'il peut exister l tel que $h(l) = \beta$ et que $\beta \notin W$.

Si $h(l)$ est définie alors $\{h(l)\} = f^{\leftarrow}(g(l)) \cap f'^{\leftarrow}(g'(l))$.

Par conséquent,

$$a_1(f(\beta)) = a_1(g(l)) = e$$
$$a'_1(f'(\beta)) = a'_1(g'(l)) = e'$$

et donc, puisque $a \xleftarrow{g} c$ et $a' \xleftarrow{g'} c$, on ne peut avoir que la situation suivante :

$$c_1(l) = e'' \wedge e'' \geq e \wedge e'' \geq e'$$

Finalement,

$$c_1(l) = e'' \wedge b_1(h(l)) = b_1(\beta) = e \sqcup e' \leq e'' \text{ par définition de la borne supérieure}$$

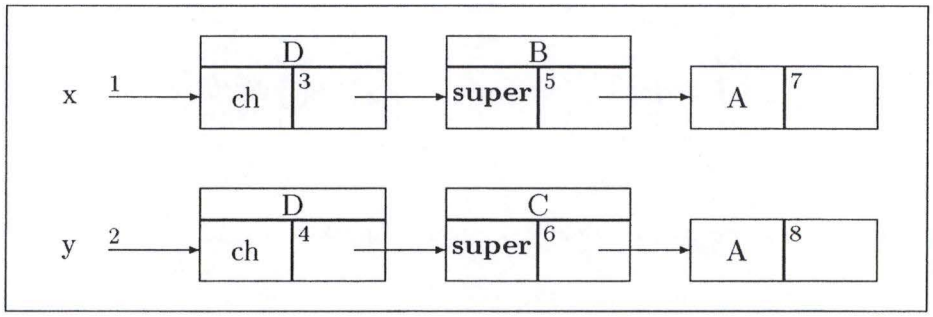

Figure 5.10: Situation $P1$

5.5 Illustration

Dans cette section on décrit un exemple simple qui met en exergue la complémentarité des deux visions du "sharing". Par complémentarité, nous entendons simplement que l'une des optiques n'est pas strictement préférable à l'autre puisqu'il existe, d'une part, des situations où la première optique donne une meilleure information et, d'autre part, des situations où c'est la seconde optique qui fournit de meilleurs résultats.

Il est évident que nous prenons ici quelques libertés par rapport à la syntaxe **SAP** (la première qualité de celle-ci n'étant certainement pas sa lisibilité).

Soit `P1` le morceau de programme suivant

```
D x = new D();
D y = new D () ;
x.ch = new B ();
y.ch = new C ()
```

et soit `P2` le morceau de programme suivant

```
D x = new D ();
D y = x ;
x.ch = new B ();
y.ch = new C ()
```

Ces deux morceaux de programme interviennent dans le contexte de déclarations suivant : le type D possède un seul champ *ch* de type A; le type C et le type B étendent le type A (nous supposerons que toutes ces classes n'ont pas de champ). Le constructeur $D()$ ne fait qu'initialiser le champ *ch* à **null**. Les constructeurs $B()$ et $C()$ ne font rien[1].

Si on exécute `P1`, on doit obtenir au niveau concret la situation $P1$ représentée dans la figure 5.10. Les locations 7 et 8 de cette figure correspondent au champ identificateur d'instance pour la classe A.

[1] On entend par là qu'il ne font que créer les instances ainsi que l'instance "père" si besoin est.

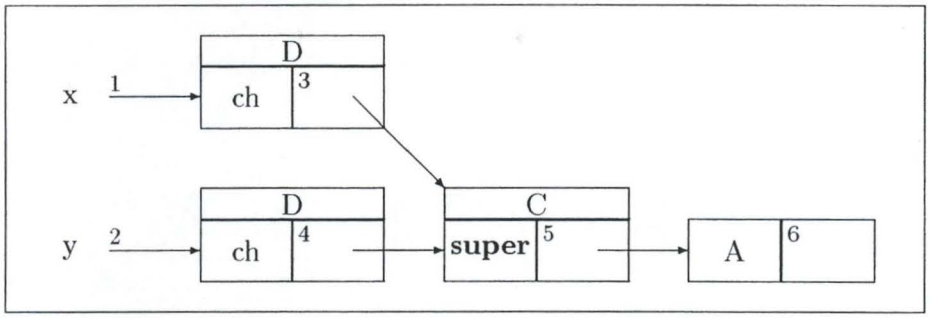

Figure 5.11: Situation $P2$

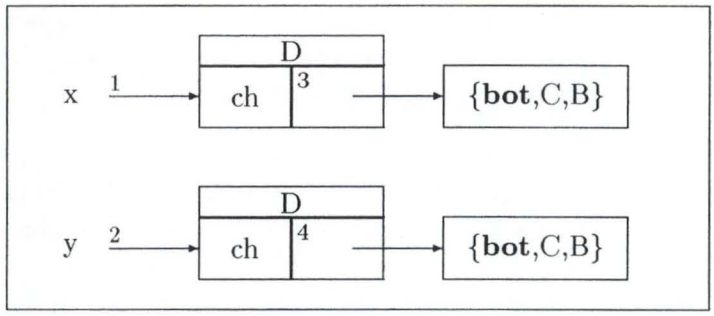

Figure 5.12: Situation abstraite après exécution de `P1` dans $(\mathbb{D}^{\#}, \leq^{*2})$

La figure 5.11 représente, quant à elle, la situation $P2$ obtenue après exécution de `P2`.

Considérons pour commencer le programme `P1`. Les premiers domaines, i.e. les domaines où le "sharing" est une perte d'information, permettent de modéliser qu'après la construction des deux instances de D désignées par x et y, ces deux instances sont distinctes. Ainsi lorsqu'on effectue "abstraitement" l'instruction `x.ch = new B()`, on sait qu'on l'effectue sur x et pas sur y. De même pour la dernière instruction. On obtient par conséquent une situation abstraite identique à la situation concrète $P1$. On conserve donc une information totale sur les types.

Dans le second domaine, une fois les objets correspondant à x et y créés, rien ne permet de dire si ces objets sont distincts. Par conséquent, on ne sait pas lorsqu'on applique un constructeur sur $x.ch$ si on ne l'applique pas également sur $y.ch$ et réciproquement. On obtient donc la situation abstraite représentée par la figure 5.12. Il est évident que, dans ce cas de figure, le premier domaine donne de meilleurs résultats.

Considérons maintenant le programme `P2`. Après application du constructeur $D()$ sur x, x et y désignent la même instance. Au niveau abstrait, on obtient un "sharing" qui, dans le premier domaine, ne fournit aucune information. Aussi, lorsqu'on applique le constructeur sur $x.ch$ on ne sait pas si on l'applique simultanément sur $y.ch$. On ne peut donc conserver pour ce champ qu'une information approximée : $\{\mathbf{bot}, B\}$. De même lorsqu'on applique le second constructeur. On obtient par conséquent la situation abstraite représentée à la figure 5.13.

Dans le second domaine, le "sharing" signifie que x et y désignent effectivement la même instance.

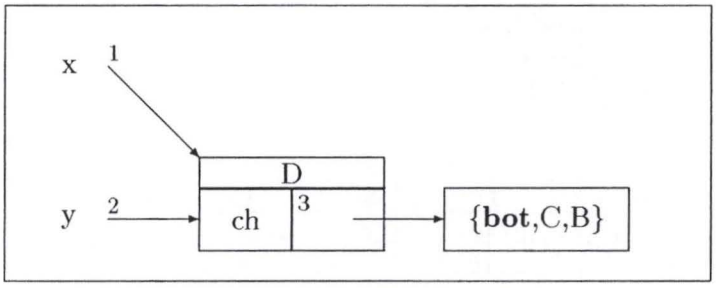

Figure 5.13: Situation abstraite après exécution de P2 dans $(\mathbb{D}^{\#}, \leq^{*1})$

Par conséquent, on ne doit effectuer aucune approximation et on obtient une situation abstraite identique à la situation concrète représentée dans la figure 5.11. Il est donc évident que, dans ce cas de figure, le second domaine donne de meilleurs résultats.

Plus généralement, on peut s'attendre, assez logiquement, à ce que le premier domaine donne de meilleurs résultats dans des situations ne présentant pas ou peu de "sharing" tandis que le second donnera de meilleurs résulats dans des situations où existent de nombreux points de partage.

Conclusion

Nous avons, dans ce travail, complètement défini un sous-ensemble représentatif du langage ciblé, à savoir Java. Nous avons d'autre part posé les premiers jalons d'un analyseur de types pour ce sous-ensemble puisque nous disposons maintenant de deux domaines abstraits pouvant soutenir celui-ci. Les problèmes techniques que nous avons rencontrés lors de l'élaboration de ces domaines nous ont, en outre, permis de mieux percevoir l'interprétation à donner à ceux-ci; ce qui est une bonne base pour la suite. Par "suite", nous entendons la définition de la sémantique abstraite et l'implémentation de l'analyseur qui sont deux étapes essentielles à l'obtention d'un premier système.

Travaux futurs

Une première liste de tâches restant à réaliser émerge d'une simple lecture de ce rapport puisque certains problèmes ont été uniquement mentionnés ou résolus de manière incomplète. Parcourons donc les différents chapitres de ce travail.

- Au cours du chapitre 2, nous avons laissé en suspend la preuve de la conservation des propriétés caractéristiques des états par les règles de transitions.
- Il est évident que le système de transitions du chapitre 3 devrait s'accompagner d'une preuve similaire. Mais plus fondamentalement, il nous faudrait prouver le théorème d'approximation de la première sémantique par la seconde que nous nous sommes ici contentés d'énoncer. Il en va naturellement de même pour le théorème d'équivalence concernant la sémantique "avec horloge".
- Au niveau théorique, la propriété 4.8 du chapitre 4 exprime la monotonie de la fonction de concrétisation par rapport au préordre. On peut alors naturellement se demander si la propriété réciproque n'est pas également valide : si un élément abstrait représente tous les éléments concrets représentés par un autre, ce premier élément abstrait est-il approximé par le second ?

  Mais les points les plus importants restant en suspend dans la présentation du premier domaine abstrait sont sans conteste les points relatifs à la borne supérieure : d'une part, il nous faut absolument rédiger et prouver l'algorithme de calcul de la borne supérieure sur $\mathbb{D}^{\#1}$ (si bien sûr celle-ci existe) et, d'autre part, il serait sans doute souhaitable de tenter de développer un algorithme "chaotique" calculant un "majorant minimal" sur $\mathbb{D}^{\#}$.

- Au niveau théorique, nous n'avons pas eu le temps de nous pencher sur la question des relations d'équivalences pour le domaine présenté au chapitre 5.

A plus long terme, on peut évidemment envisager une liste de tâches un peu plus conséquentes.

- Il serait sans doute souhaitable de "paufiner" le cadre théorique présenté. Nous avons en effet défini pour les différents domaines proposés des fonctions de concrétisation et des fonctions d'abstraction et ce de manière indépendante. Or, le cadre théorique le plus répandu pour l'interprétation abstraite reste le cadre des "insertions de Galois" dans lequel ces deux présentations se déterminent l'une l'autre. Les domaines qui permettent la définition d'une borne supérieure pourraient très bien s'intégrer dans ce cadre théorique.
- Comme nous l'avons déjà mentionné, une des étapes essentielles restant à franchir est l'élaboration de la sémantique abstraite. Cette élaboration nécessitera certainement la définition de nouvelles opérations sur les différents domaines.
- Nous avons au cours de ce travail présenté deux domaines duaux et nous avons en outre montré que ces deux domaines sont "complémentaires". A partir de là, il semble opportun de mieux cerner l'expressivité de chacun de ces domaines afin de pouvoir construire un nouveau domaine intégrant les possibilités d'expression de ces deux domaines.
- Un autre enrichissement du domaine résulterait peut-être de l'abandon de notre vision "binaire" de l'approximation des types. En effet, lorsqu'on approxime le type d'une instance parce que dans une situation cette instance est d'un certain type A et que dans une autre situation cette instance est d'un autre type B, on pourrait en fait conserver la portion de structure commune à ces deux instances (cette portion correspondrait en fait à un certain type C dont héritent A et B).
- On pourrait également remettre en question notre choix d'élimination de la pile dans les états et tenter d'approximer "directement" la première sémantique.
- Pour terminer, il est clair que si nous voulons réaliser une implémentation "effective" d'un analyseur pour Java, il nous faudra élargir la définition de notre sous-langage pour tenir compte de caractéristiques telles que les méthodes statiques et les attributs d'accessibilité.

Références

Le papier séminal sur l'interprétation abstraite est bien sûr [CC77]. On consultera aussi [CC92, Mar93] pour une présentation des diverses approches possibles de l'interprétation abstraite. L'approche utilisée dans ce travail est l'approche opérationnelle décrite dans [LC99b, LC95].

Les domaines abstraits présentés dans ce travail sont conçus selon une approche similaire à celle utilisée dans [CLCVH94, CLCVH] pour le domaine générique `Pat(`$\mathcal{R}$`)` et dans [Mus90, LCVH94] pour le domaine de substitutions abstraites `Pattern`. Précisément, l'analogie s'applique à notre domaine du chapitre 5 pour lequel la borne supérieure existe. Les mêmes idées sont aussi utilisées

dans [LCLRC99] qui contient une bibliographie complète des articles relatifs à ces domaines, tous construits pour l'analyse de programmes logiques.

Les méthodes de mise en correspondance des locations utilisées dans ce mémoire ont aussi des similitudes avec les "constraint mappings" étudiés dans [LLC96].

En ce qui concerne les aspects syntaxiques du langage, nous avons utilisées des notations similaires à celles utilisées dans [BLCLP98, BLCLP99].

Bibliographie

[BLCLP98] D. Baldan, B. Le Charlier, C. Leclère, and I. Pollet. Abstract Syntax and Typing Rules for Mercury. Technical Report RR–98–001, Institut d'Informatique, university of Namur, Namur, Belgium, December 1998.

[BLCLP99] D. Baldan, B. Le Charlier, C. Leclère, and I. Pollet. A Step Towards a Methodology for Mercury Program Construction: A Declarative Semantics for Mercury. In *Post-Proceedings of (LOPSTR'98)*, number 1559 in Lecture Notes in Computer Science. Springer Verlag, 1999.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CDG] C. Chambers, J. Dean, and D. Grove. Whole-Program Optimization of Object-Oriented Languages. Technical report, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, Washington 98195–2350 USA.

[CLCVH] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Conceptual and Software Support for Abstract Domain Design: Generic Structural Domain and Open Product. Technical Report CS–93–13, Brown University.

[CLCVH94] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming. In *Proceedings of the 21th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.

[Deu94] A. Deutsch. Interprocedural May-Alias Analysis for Pointers : Beyond k-Limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Designe and Implementation*, pages 230–241, Orlando, Florida, June 1994.

[LC95] B. Le Charlier. Abstract Interpretation and Finite Domain Symbolic Constraints. In Andreas Podelski, editor, *Constraint Programming : Basics and Trends, Proceedings of the 1994 Châtillon Spring School, Châtillon-sur-Seine, France, May 1994*,

number 910 in Lecture Notes in Computer Science, pages 147–170. Springer-Verlag, March 1995.

[LC99a] B. Le Charlier. Définition du Langage `Vas-T'y-Frotte`. Institut d'Informatique, Université de Namur, Belgique, 1999. Notes de cours.

[LC99b] B. Le Charlier. Interprétation Abstraite. Institut d'Informatique, Université de Namur, Belgique, 1999. Notes de cours.

[LCLRC99] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. Automated verification of prolog programs. *Journal of Logic Programming*, 39(1-3):3–42, April 1999.

[LCVH94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.

[LLC96] C. Leclère and B. Le Charlier. Two Dual Abstract Operations to Duplicate, Eliminate, Equalize, Introduce and Rename Place-Holders Occurring Inside Abstract Descriptions. Research Paper RP-96-028, University of Namur, Belgium, September 1996.

[Mar93] K. Marriott. Frameworks for Abstract Interpretation. *Acta Informatica*, 30:103–129, 1993.

[Mus90] K. Musumbu. *Interprétation Abstraite de Programmes Prolog.* PhD thesis, Institute of Computer Science, University of Namur, Belgium, September 1990. In French.